Lab 2 – 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 through chapter 3, and in addition section 6.1
- Event-driven programming handout
- State machines handout
- [optional] Fly24 chapter 5 – Interrupts

Concepts

- Variable Scope
- Structs
- Interrupts
- Event Driven Programming

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.
- Buttons.h – contains an enum with all the possible states of your Button Events: ButtonNoEvent, Button3UpEvent, Button3DownEvent, etc. Also includes a macro to read the current state of the buttons, as well as the two function prototypes you will implement in a Buttons.c file.

Assignment requirements

- Your program should light a single LED (D3-D8) at a time on the development board, starting in one direction and reversing upon reaching the end, in essence bouncing off the left end of the row LEDs to the D8 LED. This means that LED D9 should ONLY be lit by pressing button S4
Provided for you is the definition of a struct called "bounceSensors" to hold the counter and speed (both of type char). This struct will be used in both main() and the ISRs.

Also provided for you is the declaration of a single global variable that is an instance of the bounceSensors struct called sensors. The interrupts will be modifying this struct to store the necessary values.

Define the constants TRUE (1), FALSE (0), LEFT (anything), and RIGHT (anything not equivalent to LEFT) and use them with your while-loop and for the direction of your bouncing LED.

Only update the LEDs every 10 counter increments by checking it in main().

Map the 10-bit ADC input into 3 bits before storing it in the struct. This cannot require any addition, subtraction, division, or multiplication operations.

Use this 3-bit value generated from the ADC input to alter the timer preload register’s (PR1) upper-3 bits, so that the timer ends up triggered faster as the input gets higher. This cannot require any addition, subtraction, division, or multiplication operations.

Buttons should modify your LEDs in the following ways:

- S3 – Toggle LEDs D3 and D4 (Simultaneously) on and off.
- S6 – Toggle LEDs D5 and D6 (Simultaneously) on and off.
- S5 – Toggle LEDs D7 and D8 (Simultaneously) on and off.
- S4 – Toggle LED D9 on and off.

Note that the effects of pressing a button should be independent of the LED bouncing from right to left. If all of the LEDs are toggled on by the buttons, then the bouncing LED should still be changing positions ‘invisibly underneath’ the toggled LEDs.

Make sure that all data types, variables, macros, etc. that you code for your buttons library are not exposed in the header file. You will not be submitting your header file so any code that relies on it will not work and you will receive a 0 for this part of the lab. Additionally if this causes your code to fail to compile you will receive a 0 for the entire lab so be careful. Also, any global variables in your library should be declared as static so they are not accessible from outside the library.

Implement the Timer1 ISR and within it:

- Increment the counter member of the struct bounceSensors by 1.
- Reset Timer1’s interrupt flag at the end of the interrupt.
- Replace the multiline comment above to describe what the function does.

Implement the ADC1 ISR.
Read the current ADC value, and update the speed member of the bounceSensors struct so that it contains ONLY the most-significant 3-bits of the ADC value. Note that the ADC value is only the lowest 10 bits of the int data type it’s stored in.

Reset the interrupt flag at the end of the interrupt.

Replace the multiline comment above to describe what the function does.

- Add inline comments to explain your code. Not every line needs a comment, but instead group your code into coherent sections (like paragraphs in an English essay) and give those comments.
- 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 and Buttons.c file as comments:
  - Your name
  - 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 two lost points.
- Submit yours bounce.c and Buttons.c files (named correctly) via eCommons before the due date.

Grading

This assignment again consists of 11 points:
- 5 points – Properly implementing the interrupts
- 4 points – Making the LEDs bounce
- 2 points – Correctly implementing the Buttons library

You will lose points for the following:
- -10 points: code doesn’t compile (seriously)
- -2 points: any compiler warnings
- -2 points: failure to adhere to the style guidelines
- -2 points: the files you submit aren’t named as described in this document or you submit more than just the required documents
- -2 points: if gotos were used
- -2 points: no name or collaborators given at the top of Buttons.c and bounce.c

Program flow

The ISRs will trigger asynchronously as main() runs, updating the sensor values. Those values will be checked in the main event loop and light the LEDs appropriately. You program will probably follow the outline below:
while (TRUE)
  check buttons
    if Button 3 up event
      toggle D3 and D4 LEDs
    if Button 6 up event
      toggle D5 and D6 LEDs
    if Button 5 up event
      toggle D7 and D8 LEDs
    if Button 4 up event
      toggle D9 LED
    if counter >= 10
      if next LED should be triggered
        trigger next LED
      if we’re at the last LED
        reverse direction
  update LED display
  update the speed

Program input/output

This program uses some of the hardware features of the Proteus simulator, specifically a potentiometer, four buttons, and a row of 8 LEDs. You have used the LEDs in a previous lab, but the potentiometer and the buttons are new.

On the left is the potentiometer (in the oval) along with two arrows: up and down. These change its value and increase and decrease respectively the output of the ADC (described later). The buttons each have two possible events: down and up. The up events will be used to signal the release of a button and the desire to toggle certain LEDs.
Event Driven Programming

The provided pseudo code for the program flow shows the use of a very powerful approach to programming known as Event-driven Programming. This lab introduces you to the concepts of system state and event checking.

A state is the present condition of a system or entity. For example, a car has many states: such as what gear it’s in, its speed, the position of the driver’s seat, etc. Buttons have states, though they’re simpler, up and down. Notice that these completely define the button. Systems use states to determine what actions they should undertake. For example, when we’re in the MOVING_LEFT state, the next time the LED should move it will move to the left.

Systems can also transition between states using either external triggers (such as the timer interrupt that we will use in this lab) or via other properties of the system. For instance, if you are moving to the right and are at the last LED then you will want to transition to the state of moving to the left.

Within the examples of states above, event checking has already been implied. Event checking involves testing for changes in the triggers that the program watches: Has button S3 been pressed? Has the counter reached 10?

Following the event checking code is the transition code that checks if a state transition should occur given the most recent events and if so what that should entail. Like the example above, when your direction is to the right and the last LED is already lit then it is time to update your direction state.

Buttons

In this lab you are required to use four buttons (S3-S6, note that they are not in numerical order, this detail will become important when matching which buttons toggle which LEDs). The functionality of these buttons requires some knowledge of the PIC24f series itself and relies on code defined in the p24fxxx.h header file that is included for you in Buttons.h

The buttons library will need a static variable of type char to store the last button state in order to compare this value to the current button state.

The InitButtons() function will have to enable the D6/7/31 and A7 pins as inputs for the micro processor. This is done with the following two statements:

```
TRISD |= 0x20C0;
TRISA |= 0x0080;
```

Also, InitButtons() will need to initialize the last button state to hold the 8-bits of data returned by READ_STATE() (which should be stored in a static variable of type ‘char’). Remember that this function must be called at the start of your program before CheckButtonEvents() can be used.
The CheckButtonEvents() function will handle the logic to signal a transition in events. This means that when this function is called it should read in the current button state from the READ_STATE() macro provided for you in Buttons.h, compare this value to the last button state, and if they are not equivalent determine what events have been triggered. Deciding what to do depends on the transition you have taken. For example, if the last button state was Button3Up and the current state is not Button3Up then the event that has occurred is the Button3Down event. Notice that more than one button event can occur at a time. You should therefore use bit masking to set the event variable that is returned by CheckButtonEvents().

Interrupts

Interrupts are a way for a processing to handle a time-sensitive event. When an interrupt occurs whatever code the processor had been executing is paused and it shifts over to executing something special code defined just for this event generically called an Interrupt Service Routine (ISR). ISRs should be written so that when the interrupt is called the processor can quickly handle it and then get back to normal execution of the program. This means that expensive operations such as printf() or user prompts SHOULD NOT be done within interrupts as it will affect normal program execution. Interrupts can also occur at any time and so your code can be interrupted at any time.

A special case occurs when the processor is inside of an ISR and another interrupt is triggered. The processor uses interrupt flags to keep track of when it is inside of an interrupt. These flags must be cleared (set to 0) at the end of every interrupt so that they and other interrupts can trigger.

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, like receiving data over a serial port.

Interrupt Service Routines look very similar to functions, though they have specific names and include some additional annotations that you may safely ignore. For the PIC24F ISRs are declared as follows, with INTERRUPT_NAME replaced by the actual interrupt name such as "T1Interrupt":

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

Note that it returns nothing and takes no arguments. Also, an interrupt should not deal with any function-level static variables. Since your code doesn't call these functions directly, there is no way to pass arguments or process the return value. The way to get data into and out of an ISR is to declare variables at the module-level: outside of any functions or curly-braces (look for the comment above main()). Since interrupts need to be very short, you should ONLY be using your ISR's to store values from the predefined registers into the bounceSensors struct.

LEDs

The LEDs D3-D10 are controlled by configuring pins on the processor: setting these pins high turns them on and low turns them off. These LEDs are all connected to the A-pins.
and so are controlled by either the LATA register or the LATAbits struct. The LATA register is used to configure multiple 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 these control the LED; this means that you shouldn’t be setting LATA to anything above 255.

```c
// Turn on the LEDs connected to RA0 and RA1
LATA = 0x3;
```

Note that hexadecimal is commonly used for bit-fields because it’s very easy to interpret a hexadecimal number as binary after a little bit of practice.

**Configuring individually:**

LATAbits is a struct with members that look like LATA0, LATA1, etc. that correspond to pins A0, A1, etc. You can refer to its specific members like so:

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

**Timers**

Timers trigger an interrupt repeatedly at a programmed frequency. These are 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 the bounceSensors counter member. Timer1’s interrupt flag is IFS0bits.T1IF, and this bit should be cleared at the end of the ISR.

The timer is configured to apply a prescalar to the processor frequency to slow it down (in turn changing the frequency at which the interrupt is called). Timer1’s prescalar is set by the register PR1, with a valid range from 1 to 65536 (or 0xFFFF) with 1 being the highest frequency/fastest interrupt rate. 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. Account for this within main() the pseudo code above does in the Program flow section of this lab manual.

**Analog to digital converters (ADC)**

Think of a volume dial, an input with infinite settings between a minimum and maximum. This is what’s called an analog control: as it has a large range of possible values. This compares to a digital control, which has only 2 possible values. Hardware called an analog-to-digital converter is used to translate these analog signals into digital values that a processor can understand in the form of 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.

Your job is to map this 10-bit value into a 16 bit value that is used by the PR1 register used as a prescalar for Timer1. More information about integer range mapping is given below, but you will have to keep the following in mind:

- The value stored into ADC1BUF0 is a 10-bit value that stores the current value of the ADC that you are working with.
- Since your interrupt service routine needs to be short you will only save the three most significant bits into the 8-bit speed member of your struct.
- The PR1 register holds a 16-bit value which controls the frequency at which the Timer1 ISR will be called (so higher values will call the ISR less frequently than lower values).
- The PR1 register should contain 1 of only 8 significant values, all of which have equal distances from each other on a number line.

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

**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 can cause performance problems if relied on exclusively. 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 value of the integer? A left bit-shift moves all the bits up 1 bit to higher-valued bits, which is the same as multiplying by 2. Shift more to multiply by higher powers of two: so bit-shifting 4 times would multiply the integer by $2^4$ or 16. A similar process holds for bit-shifting right, it just acts as 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 0x0A, which are just the highest 4 bits.
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. The reason we don’t just use the lower-order bits is because we want the lowest values in each range to correspond to each other and the highest ones to also correspond, and to be increasing for both. If the lowest are used we would see something like the following picture. The black line shows a proper mapping of the range \([0, 1023]\) to \([0, 31]\) while the red shows the improper mapping described above.

Usually when mapping integer ranges it is the lower-order bits that get dropped as the higher-order bits contain “more” data as they represent large values of the number (the 5th bit of a number represents 16 versus the 1st bit represents 2, so if the 1st bit was “lost” the resultant number would be closer to the actual number versus if the 5th bit was “lost”).

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. Many operations will often need to be combined to get the desired result which sometimes 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
**Incremental Development Approach for this Lab**

Here is a logical way to approach this lab:

1. Light up one LED
2. Implement Timer1’s ISR
3. Every 10 timer interrupts move the LED one position (from one end to the other).
4. Add bouncing to the LED movement.
5. Implement the ADC ISR.
   - Test by lighting up D10 if ADC > 500
6. Implement InitButtons().
   - Test by lighting up a specific LED based on which button is pressed.
7. Add CheckButtonEvents().
8. Implement the required Button functionality for the lab.
9. Double-check your code and comments before submission.

Remember that you will be required to make a incremental development plan for all future labs. These will not be graded, but if you seek help from a TA in person you will be required to show this plan, explain it, and indicate where you currently are in its progress.