UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 6 - Bouncing LEDs

Commit ID Form: https://goo.gl/forms/3ufpo5ErXmzA0MV83

15 Points

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. Additionally you will start to use the various hardware on the I/O Shield: the OLED, potentiometer, switches, buttons, and LEDs. This lab highlights two typical applications: reading sensors and triggering output.

Reading

- **K&R** – Chapters 1-3, 4.11.2, 6.1
- **CKO** – Chapters 5.0 – 5.7
- Bit manipulation handout

Concepts

- Variable Scope
- Structs
- Macros
- Interrupts
- Hardware: LEDs, OLED, buttons, and switches
- Software button debouncing
- Bit manipulation/bit masks
- Event-driven programming

Provided files

- bounce_part1.c, bounce_part2.c, bounce_part3.c – contains `main()` and other setup code for each of the separate parts of this lab.
- bounce_ec.c – contains `main()` and other setup code for doing the extra credit
- Buttons.h – You will implement the corresponding Buttons.c file that lets you poll for button events. It contains an enum with all the possible states of your

Button Events: `BUTTON_EVENT_NONE`, `BUTTON_EVENT_3UP`, `BUTTON_EVENT_3DOWN`, etc.

- BOARD.c/h - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values. **You will not be modifying these files at all!**
- Oled.c/h, Ascii.c/h, OledDriver.c/h – These files provides all the code necessary for calling the functions in Oled. You will only need to use the functions in Oled.h, the other files are called from within Oled Library. **You will not be modifying these files at all!**

Assignment requirements

Part 1:

- Implement the Leds.h header library (there is no Leds.c)
  - See the LEDs and Macros section.
  - Add comments with your name and any collaborators.
  - Add a header guard using the LEDS_H constant.
  - Add an "#include" to include the Microchip's xc.h header (so use angle-brackets), which will define registers you will use later in this file. Don't forget to include BOARD.h above this include to ensure proper functionality
  - Implement the LEDS_INIT() macro that sets the TRISE and LATE registers to 0. See the section here on macros and implement this in do-while form. Don't add a semicolon at the end of this macro!
  - Implement a LEDS_GET() macro that returns the value of the LATE register. Don't add a semicolon within the macro!
  - Implement an LEDS_SET(x) macro that sets the LATE register to x. Be sure to surround x in parenthesis in the macro. Also, don't add a semicolon within the macro!
  - You will use this file in later labs, we will never provide this for you, so make sure you get this working properly for this lab and save it!
- Your program should light a single LED at a time on the development board. This LED will "move", starting in one direction and reversing direction upon reaching the end, in essence bouncing back and forth between the LD1 and LD8 LEDs.
  - Define a struct datatype called "TimerResult". The TimerResult struct should have a uint8_t member named "event" and a unsigned 8-bit variable named "value".
  - Define a module-level variable for storing your timer event data, of type "struct TimerResult". These will be used for persistent data storage in the interrupt and for checking for events in main().

    - Define the constants LEFT (value: 1) and RIGHT (value: 0) at the top of bounce.c and use them within your while-loop and for the direction of your bouncing LED.
    - The speed of the LED should be controllable by the settings of the 4 switches on the I/O shield. The lit LED should bounce slower the more switches are up (closer to the OLED). This is done by changing when the Timer1 interrupt sets that a timer event has occurred. Only every i^{th} timer interrupt should actually trigger a Timer1 event, so only set the timer event flag when the counter has exceeded the value returned by `SWITCH_STATES()` (see BOARD.h).
- Implement the Timer1 ISR and within it:
    - Increment the `value` member of your struct TimerResult variable by 1.
    - If `value` ends up being greater than the value returned by the `SWITCH_STATES()` macro, set the `event` member to `true`. Also reset `value` to 0 if that is the case.

Part 2:

- Define a struct datatype called “AdcResult”. The AdcResult struct should have a uint8_t member named "event" and an unsigned 16-bit value named "value".
- Define a module-level variable for storing your ADC event data of type "struct AdcResult". These will be used for persistent data storage in the interrupt and for checking for events in main().
- Implement the ADC1 ISR
    - Read the 8 buffered ADC values in the ADC1BUF0 through ADC1BUF7 variables and average all of them. If the resultant value is different than the last averaged reading of the ADC, store it in the `value` member of the AdcResult struct and set the `event` member to `true`.
- Use the functions in Oled.h (OledDrawString() specifically) to display the current potentiometer value (this should be the averaged ADC value, so between 0 and 1023) and what percentage of the total that equates to. This should output a complete 0-100% range. Note: You will need the Oled files for this to work.

Part 3:

- Implement the Buttons library in a Buttons.c file
    - Add comments for your name and any collaborators.
    - Implement all of the functions described in Buttons.h. You'll want to write test code to check this similar to what was done for Matrix Math.
    - 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.
    - The ONLY way to use the Buttons library is to use the functions and datatypes we'd provided in the Buttons.h file. Doing anything else will result in your code failing compilation and you getting a 0.
- Use the four buttons to toggle the state of the 8 LEDs in groups of 2:
    - BTN1 - Toggles LD1 and LD2
    - BTN2 - Toggles LD3 and LD4
    - BTN3 - Toggles LD5 and LD6
    - BTN4 - Toggles LD7 and LD8
- For each button, either its UP or its DOWN event should actually toggle the LED states. This will be dependent on the state of the four switches on the I/O Shield. If a switch is in the up position (moved closest to the OLED), then the corresponding button's UP event will trigger the LED change. So if SW3 is up, then pressing and holding BTN3 will do nothing and only on releasing it will the LEDs toggle.

General:

- Document your code! 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 essay, and give those comments.
- 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 in libraries will result in no credit for that section. Compilation errors in the main file will result in NO CREDIT. Any compilation warnings will result in two lost points.
- **Specify a lab partner for BattleBoats:** Lab 9, BattleBoats, is a partnered lab. For this lab you are required to specify your BattleBoats partner in your README.txt file and fill out the google form found here. Make sure that they have specified you as well, or you might not be able to get them. Specifying a partner in your readme and on the form will count for 1 point for your BattleBoats grade so be sure to do this. If you do not specify a lab partner, you will be randomly assigned one!
- **Extra credit:** Combine parts 1 through 3 into a single program in bounce_ec.c with the following modifications:
    - The switches only change when the buttons affect the LEDs.
    - The speed of the LED is now controlled by the pot (read the adcData.value member directly in your timer interrupt), with the top 4 bits of the ADC value controlling when the timer triggers an event, resulting in the values 0-15 being used here. This is the same range that

were used before so the LED should bounce at the same speeds as it did in Part 1.
    - The LED should bounce faster as the pot is rotated clockwise.
- Create a readme file named README.txt containing the following items. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.
    - First you should list your name & the names of colleagues who you have collaborated with.
        - NOTE: collaboration != copying
    - **NEW!** Specify your lab partner for Lab 9, Battleboats. This will count for 1 point for that lab. If you do not specify a lab partner, you will be randomly assigned one!
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? How did you work with other students in the class and what did you find helpful/unhelpful?
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help your understand this lab or would more teaching on the concepts in this lab help?
- **Required Files**
    - bounce_part1.c
    - bounce_part2.c
    - bounce_part3.c
    - Buttons.c
    - README.txt
    - Leds.h

Grading

Grading for this lab will be different in that your source files will not necessarily be compiled together. Since part of this assignment entails the creation of libraries, those will be tested independently of your bounce.c file and instead with a program written to test your libraries' functionality. If your libraries aren't properly self-contained or significantly deviate from the specifications you've been given, we may encounter a compilation error when testing. This would result in a significant loss of points. An easy way to prevent this is to only use the libraries in accordance with their descriptions and the functions & variables defined in the header. If you have concerns please ask BEFORE the assignment has been submitted.

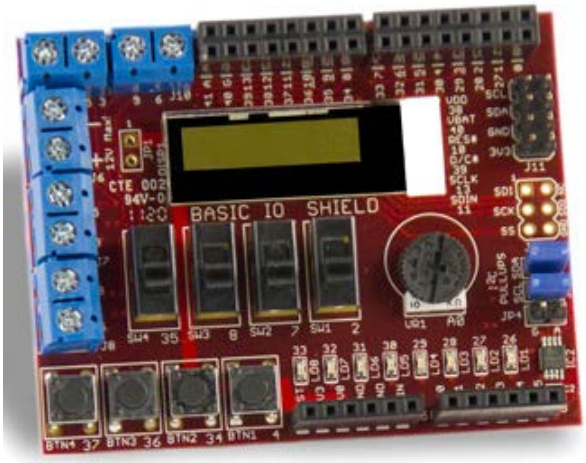
This assignment consists of 15 points:

- 3.25 points - Part 1
- 2.25 points – Part 2
- 1.75 points - Part 3
- 2.75 points – Correctly implementing the Leds library
- 3 points – Correctly implementing the Buttons library
- 1 point – Style guidelines were followed with < 10 errors
- 1 point - README.txt
- **Extra Credit:** 1 point for combining all 3 parts into bounce_ec.c with slight modifications.

You will lose points for the following:

- If a source file doesn't compile, you'll lose all points that rely on that file
- -2 points: any compiler warnings
- -2 points: if gotos, externs, or global variables were used

Program input/output



On the Basic I/O Shield shown above we have all the hardware used for this lab. The round black circle on the right is the potentiometer. Rotating it clockwise increases the voltage sensed by an ADC pin, while counter clockwise rotation does the opposite (see the ADC section for more details). The four push buttons are the silver squares along the bottom left, numbered BTN1 through BTN4, starting from the right. The OLED is a monochrome display, shown as the green rectangle towards the top of the shield, that can display raw pixels or text characters. The four switches are the vertical black rectangles between the buttons and the OLED, and named SW1 through SW4. The LEDs are the white rectangles along the bottom, named LD1 on the right through LD8 on the far left.

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.

State is just a term used to refer to a specific aspect 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 generally have a single state: pressed, which can be true or false. Note that this state completely defines 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 its limit?

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.

Bit masks

When working with bits it's common to use a "bit mask". A bit mask is a filter used when modifying a value such that only certain bits are affected. In fact this concept was already introduced in the **Buttons** section with: `TRISD |= 0x20C0;`

What that statement does is mask on the 6th, 7th, and 13th bits in the `TRISD` register by OR-ing it with `0x20C0`. While in this case the bit mask involved setting the bits (to 1) by using an OR operation, bit-masking can also refer to clearing bits (to 0) or toggling bits (changing their value) by using AND or XOR operations respectively.

In the case of this lab masking will come in with Buttons library with the return value of `ButtonsCheckEvents()`. Since more than one event can trigger in a given call to `ButtonsCheckEvents()`, the return value will need to be masked for every event to check if it occurred independently of any other event.

This bit masking will also come into play in the extra credit, where you have a bit mask of which LEDs should be on based on both the bouncing LED and the LEDs set by the buttons. You will need to combine both of these masks to obtain the final output of what the LEDs should be set to.

Buttons

In this lab you are required to use four buttons named BTN1 through BTN4. The functionality of these buttons requires some knowledge of the PIC32 series itself and relies on code defined in the xc.h header file that is included for you in Buttons.h.

The `ButtonsInit()` function will have to enable the D5/6/7 and F1 pins as inputs for the microprocessor. This is done with the following two statements:

```
TRISD |= 0x00E0;
TRISF |= 0x0002;
```

The `ButtonsCheckEvents()` function will handle the logic to signal a transition in events. This function relies on reading the current button state using the `BUTTON_STATES()` macro provided for you in BOARD.h. This function tracks the current button state independently of the hardware and waits until 4 samples of the buttons all indicate a change before triggering the corresponding event. So over 5 calls to ButtonsCheckEvents() if it BTN3 was unpressed in the first call, but pressed in the next 4 calls and the last BTN3 event was BUTTON_EVENT_3DOWN, then this 5th call to `ButtonsCheckEvents()` will return a BUTTON_EVENT_3UP. This is referred to as software button debouncing, specifically 4-sample debouncing.

Note that more than one button event can occur at a time, which is why all of the button events are mutually-exclusive bits, therefore they can all be ORed together. So your library should be able to detect events independently per button, so a BTN3 up event and BTN4 down event can both occur in the same call to `ButtonsCheckEvents()`. You should therefore use bit masking to set the event variable that is returned by `ButtonsCheckEvents()`. We do test this functionality when grading, so be sure to make sure you implement this correctly!

Some example test code for the Buttons library is provided below. Make sure you understand what the expected output is from the test before you code it. Tests don't help if you don't check that your program is running them properly!

```
// The following code assumes that  ButtonCheckEvents() is called in the timer
// interrupt and storing the result in the module-level uint8_t buttonEvents.
while (true) {
      if (buttonEvents) {
            if (buttonEvents & BUTTON_EVENT_1UP) {
                  puts("BUTTON_EVENT_1UP");
            }
            if (buttonEvents & BUTTON_EVENT_1DOWN) {
                  puts("BUTTON_EVENT_1DOWN");
            }
            buttonEvents = BUTTON_EVENT_NONE;
      }
}
```

Macros

Macros are a very powerful tool that can make code simpler to implement, easier to understand, and faster (not necessarily all 3 everytime!). But with great power, comes great responsibility. The macro system in C (unlike some more modern languages) does not do any input checking and mostly does straight textual substitution. This makes it very easy to run into problems.

The most basic usage for macros are for declaring constants using the `#define` preprocessor directive like:

```
#define TRUE 1
```

Using a constant like this makes your code easier to read, so instead of a number which you may not understand, you have a name that hopefully hints to its function. The other advantage to this is that, unlike using the `const` modifier, no memory space is used to store this number, it's merely substituted in for the name wherever it appears in the source code.

Now using macro constants is the most basic usage of macros. A more advanced use is for writing multiple-statement blocks. Now these aren't functions in the regular sense, they don't return anything, but can take arguments. What they are is really a smarter way to do a straight text replacement, basically templated textual replacement.

For example, here's a macro that just divides the PR2 by a number:

```
#define DIVIDE_IT(num) (PR2 = PR2 / (num))
```

So if you call it like follows:

```
DIVIDE_IT(5);
```

Everything will compile properly and the resulting code would be:

```
(PR2 = PR2 / (5));
```

So whatever text is passed in as the argument is substituted for "num" in the output text. Also note that we included parenthesis around num. This is to make sure the following works correctly: `DIVIDE_IT(5 + 12*x);` If we didn't include the parenthesis, the text output would be, which is not what we intended:

```
(PR2 = PR2 / 5 + 12*x);
```

We also didn't include a semicolon in the macro because we want the user to add one as they normally do after regular statements in C and for the user to understand that they can treat the macro as a single statement, much like a conventional C function call. But with multiple statements in a macro we have to do something a little extra to keep that functionality. We instead use a do-while loop which results in functionality nearly identical to a function call (the "\" are to continue the macro on the next line):

```
#define INIT_ECAN(bsize) do { \
      CB_Init(&ecan1TxCBuffer, txDataArray, bsize); \
      CB_Init(&ecan1RxCBuffer, rxDataArray, bsize); \
} while (0)
```

So with that macro written using a do-while, we can call it like a single function call:

```
INIT_ECAN(10);
```

And now the semi-colon works properly and the code is grouped into a single statement.

Working with macros can be quite difficult in a language like C, where it only offers straight text replacement. This can lead to weird syntax errors that are vague and almost impossible to figure out the source of. To help debug issues which you may think are related to macros, there is a special view of the code that has all the macros expanded and inlined in MPLAB X. To view this, right click anywhere on your code and select Navigate -> View Macro Expansion. The window that opens at the bottom shows the same portion of the code that you have shown in the main coding window, but with all the macros expanded and inlined.

LEDs

The LEDs LD1-LD8 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 E-pins and so are controlled by the `LATE` register (a uint16 where the top 8 bits don't actually exist) where each bit in this register controls the output of a pin: 1 means high and 0

means low. The LEDs are on pins E0 through E7 and a high output corresponds to the LED being enabled or on. Remember that bit-numbering starts at `0`!

Enabling the proper E-pins to be outputs so that they can control the LEDs is done with the TRISE register (like the Buttons library did). Setting a bit (to 1) in the TRISE register turns that pin into an input and clearing that bit (to 0) sets it to an output. For the LEDs we want to control, the corresponding pins (and therefore bits in TRISE) should be set to outputs and therefore a `0`.

The following code snippet enabled LEDs D1 and D4 and turns them both on:

```
// Turn on the LEDs LD1 and LD4
TRISE = 0xF6;
LATE = 0x09;
```

And the library you will implement in Leds.h will implement 3 separate macros that you will use for the remainder of the labs in this class, so be sure that they work correctly. You should perform some unit testing on this library before you move on to writing any other code. Below is a single example of a test case for the LEDs. You should test the LEDs, especially the LEDS_GET() macro more extensively than with just the following test case, but this is a good place to start.

Example test for LEDs:

```
LEDS_INIT()
LEDS_SET(0xCC);
int i;
for (i = 0; i < 10000000; ++i);
LEDS_SET(0xDD);
for (i = 0; i < 10000000; ++i);
LEDS_SET(0);
for (i = 0; i < 10000000; ++i);
LEDS_SET(0xFF);
```

Interrupts

Interrupts are a way for a processor 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 input or output, 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 regularly handle interrupts, as they are generally a part of normal operation. Some interrupts involve the internal operation of the CPU while others are triggered externally, like when the CPU receives data over a serial port.

ISRs look very similar to functions, though they have specific names and include some additional annotations that you may safely ignore. For the PIC32, ISRs are just regular C functions with special annotations before the function name that indicate what ISR it implements. For example the following code is the function prototype for the Timer 1 ISR:

```
void __ISR(_TIMER_1_VECTOR, ipl4) Timer1Handler(void);
```

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 directly. The way to work around this and get data into and out of an ISR is to declare variables at the module-level, declaring them static outside of any functions in bounce.c (look for the comment above `main()`).

Since interrupts halt normal code execution, they should be short so that the main program code can continue execution. You should avoid complex work, like calling `sprintf()` or other very slow function calls. For the code you will writing in the Timer1 and ADC1 ISRs, you will only use simple comparisons and arithmetic operators.

Timers

Timers are hardware counters that trigger an interrupt repeatedly at a programmed frequency. These are commonly used for triggering output repeatedly, like they are used in this lab. In this lab you will use the 16-bit timer1 (there are several), which we have already configured for you. Whenever Timer1 triggers the `Timer1Handler()` interrupt is called. In this ISR you should increment the `timerData.value` member. The interrupt flag for Timer1 also needs to be cleared, which should be done for you as the first operation within it. If this flag isn't cleared, the processor will never leave return from the interrupt.

The timer is configured to clock at 1/8th the speed of the peripheral clock in the PIC32 series of processors. We have configured the peripheral clock to run at 20MHz, so the timer ticks at 2.5MHz. Now we have configured the trigger for when the timer initiates an interrupt to occur at the maximum it can be, 65535 or UINT16_MAX, since this is only a 16-bit timer. So the timer we have provided actually triggers an interrupt at 2.5MHz / 65535 or about 38Hz. So bouncing at the base rate of the timer interrupt will change the lit LED 38 times a second.

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 8 times, storing the resulting values as unsigned 10-bit values in the integer registers `ADC1BUF0` through `ADC1BUF7` before the ISR `AdcHandler()` is called. The interrupt flag for the ADC also needs to be cleared, which should be done for you as the first operation within it. If this flag isn't cleared, the processor will never leave return from the interrupt.

The code that you will add to `AdcHandler()` will:

- Average the 8 values stored in the variables ADC1BUF0, ADC1BUF1, ..., ADC1BUF7. These are all 10-bit unsigned integers that correspond to the voltage of the potentiometer. In fact the units of this integer are 3.3V / 1023, or 3.23mV.
- If this averaged ADC value is different from the averaged value calculated during the last interrupt, trigger an ADC event by setting the "event" member of your ADC data struct to true.
- And store the new averaged ADC value in the "value" member of your ADC data struct.

Program flow

Part 1:

The Timer1 interrupt is called continuously as `main()` runs, and it is there that you will increment a counter and check its value against the switches to determine when a timer event occurs. This event is then checked for in the event loop in `main()` to move the bouncing LED. It will follow the outline below:

```
initialize leds library
while true:
      if the timer event flag is set:
            if we're at the last LED:
                  reverse direction
                  trigger next LED
            else:
                  trigger next LED
```

```
            clear the timer event flag
```

Part 2:

The ADC interrupt is called continuously as `main()` runs, and it is there that you will average the sampled ADC values and only save the value and trigger an ADC interrupt if it is different from the last sampled value. This event is then checked for in the event loop in `main()` to move the bouncing LED. It will follow the outline below:

```
while true:
      if the ADC event flag is set:
            update the OLED with the ADC percentage and raw value

            clear the ADC event flag
```

Part 3:

The Timer1 interrupt is called continuously as `main()` runs, and it is here that you will run the button event checker and save any events it detects. This makes the sampling of the buttons happen at a regular pace, which wouldn't happen if we put the checking in our event loop. Any resulting button events are then checked in the event loop in `main()` to move the bouncing LED. It will follow the outline below:

```
initialize buttons library
while true:
      if a button event flag is set:
            store the current switch positions

            if Switch 1 is on and Button 1 up event:
                  toggle LD1 and LD2 LEDs in bitmask
            else if Switch 1 is off and Button 1 down event:
                  toggle LD1 and LD2 LEDs in bitmask

            if Switch 2 is on and Button 2 up event:
                  toggle LD3 and LD4 LEDs in bitmask
            else if Switch 2 is off and Button 2 down event:
                  toggle LD3 and LD4 LEDs in bitmask

            if Switch 3 is on and Button 3 up event:
                  toggle LD5 and LD6 LEDs in bitmask
            else if Switch 3 is off and Button 3 down event:
                  toggle LD5 and LD6 LEDs in bitmask

            if Switch 4 is on and Button 4 up event:
                  toggle LD7 and LD8 LEDs in bitmask
            else if Switch 4 is off and Button 4 down event:
                  toggle LD7 and LD8 LEDs in bitmask
```

```
clear button event flag
```

Approaching this lab

For this lab we've introduced several new concepts (interrupts, event-driven programming, etc) and a multitude of real hardware to interface with. Therefore to complete this lab it will be important to take small steps, checking your work as you go, so you don't get overwhelmed. We suggest the following steps for completing this lab.

Remember when creating a new project to turn on the “Additional warnings” option (see the Lab 1 manual for details).

Part 1 - Step 1

- Start with Part 1.
- Create the Leds.h file with header guards and implement LEDS_INIT() and LEDS_SET().
- Test by running the example test in the LEDs section of this manual.
- Add any additional testing that you think of.

Part 1 - Step 2

- Implement LEDS_GET().
- Test by setting the LEDs to a value, putting in a long pause by doing "`int i; for (i = 0; i < 10000000; ++i);`", and then using LEDS_GET() and making sure you get the same result. Do 4 different sets of these like:

  // Set LEDs to 0xFFFF
  // Long pause
  // Get LED values, compare to 0xFFFF
  // Set LEDs to 0x03
  // Long pause
  // Get LED values, compare to 0x03
  ...

  You should only use for-loops like this to delay when testing, not for the actual lab code. Your final program should NOT use any loops of any kind for delays.

Part 1 - Step 3

- Implement the Timer1 interrupt to trigger an interrupt every 10th interrupt (using the counter)
- In main() create your event loop checking or a timer event.
- If a timer event has been detected, toggle LD1 using your Leds.h library.
- Also clear the event flag in your loop.
- If this works, move on to the next stage

Part 1 - Step 4

- Now that your timer interrupt code is working, add the logic to bounce the LED.
- You’ll need to create a new variable to store the direction the LEDs are bouncing.
  - Be sure to create your LEFT and RIGHT macro constants here!

Part 1 - Step 5

- Modify your Timer1 interrupt to use the switch values as its limit on the counter to determine when a timer event occurs.

- SW4 should slow down the LED the most and SW1 the least when they're changed.

Part 2 - Step 1

- Create the struct and variable for storing your ADC event data.
- In your ADC interrupt, average all the ADC1BUFx variables, and save the resulting value and set an ADC event.
- In main(), create your event loop that checks for ADC events.
- In your ADC event checker, update the OLED with the current ADC value using `sprintf()` and `OledDrawString()`.

Part 2 - Step 2

- Add a percentage calculation to the OLED output.

Part 2 - Step 3

- Add a check in the ADC interrupt so that ADC events are only triggered if the averaged ADC value has changed since the last ADC interrupt.

Part 3 - Step 1

- Implement `ButtonsInit()`
- Create an infinite loop in main where you call BUTTON_STATES(), saving it's result to a variable and write this result to the LEDs.
- To test, run your code and pressing a button should light up the corresponding LEDs.
- Implement `ButtonsCheckEvents()` WITHOUT the software debouncing (so only use the previous sample to determine if things have changed.
- Run `ButtonsCheckEvents()` in your interrupt and write a test program following the example test code in the Buttons section in `main()` to output over UART whenever a button event is triggered. Be sure to expand this test to cover all 8 events (2 per button).
- Test by pressing the buttons and watching the serial output.

Part 3 - Step 2

- Implement software debouncing (the current button state and the last 3 button states should all be the same, and also different from the oldest button state).
- Playing with the buttons again should reveal that the flickering or sometimes missed presses are now gone.

Part 3 - Step 3

- Change the test code you had been using to instead of writing output to the serial port, toggle the LEDs.
- Confirm this is working by testing with the buttons.

Part 3 - Step 4

- Modify your button event checker to switch between checking for UP events and DOWN events for actually toggling the events based on the switch value for a given button. Remember that you'll need to do bit masking like `SWITCH_STATES() & SWITCH_STATE_SW1`

Final step

- Double-check that you've met all of the lab requirements and finish any that are still outstanding.

- Make sure your code is properly formatted, commented, and there are no compilation warnings/errors.
- Create your README.txt
- Submit all of your code and your README.txt file.

Extra credit

- Merge all of the code into bounce_ec.c and modify to fulfill the requirements for extra credit.