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

ECE-013E: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 6 – Basic IO on a Microcontroller
19 Points

Introduction

Nearly all computers are embedded devices. Approximately 40 billion (with a B) embedded devices were manufactured and shipped in 2017 alone, and you probably have several on your person right now. They manage complicated systems like your phone, or simple systems like the SD card reader in your laptop.

Embedded systems have their own special issues that designers must consider: They interact directly with hardware, they often have very limited memory (ROM/RAM), and they often must react very quickly to a wide variety of unpredictable events. They also have capabilities that are often missing in larger CPUs. In short, they are programmed in C just like the big machines, but have specifics that need to be dealt with correctly.

In this lab, you will start to work with microcontrollers in C.[1] You will learn about timers and interrupt service routines (ISRs or interrupts for short), which are used heavily in computer systems. Additionally you will start to use some of the hardware on the I/O Shield: the OLED, potentiometer, switches, buttons, and LEDs. You will learn to filter the potentiometer's readings to ensure a clean input.

This lab highlights two typical applications: reading sensors and manipulating output.

Reading

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

[1] In fact, you have been programming a microcontroller the entire time, but now we will be using the actual hardware capabilities of the embedded processor and having your code interact with the physical world. This is a small introduction into the field of embedded programming.

Concepts

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

Required files

- Leds_Lab06.c
- bounce_timers.c
- bounce_switch.c
- bounce_adc.c
- Buttons.c
- ButtonsTest.c
- bounce_buttons.c
- bounce_ec.c (extra credit)
- README.md

Lab Files:

- **DO NOT edit** these files:
  - Buttons.h – The header file for the button event checker. Contains the prototype for the event checking function, and enums to describe possible button events: BUTTON_EVENT_NONE, BUTTON_EVENT_3UP, BUTTON_EVENT_1DN, etc.
  - BOARD.c/.h – Standard hardware library for CSE13E.
  - 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.
- **Create** these files:
  - Leds_Lab06.c – a simple file to implement the functions prototyped in Leds_Lab06.h. You should also write a test harness for this file, but it is not collected as part of this assignment.
  - Buttons.c – Implement the buttons function prototyped in Buttons.h

- **Edit** these files (these are provided as *_template.c files):
  - bounce_timers.c
  - bounce_switch.c
  - bounce_adc.c
  - ButtonsTest.c
  - bounce_buttons.c
  - bounce_ec.c

Assignment requirements

Part 1 – timers and LEDs:

- **bounce_timers.c:**
  - Your program should run in a reactive way, using Timer events to monitor 3 timers.
    - Define a struct datatype called "Timer". The Timer struct should have a uint8_t member named "event" and an int16_t member named "timeRemaining".
    - Use three module-level (ie, static) instance of this struct to pass information between the Timer1 Interrupt Service Routine. Call them TimerA, TimerB, and TimerC.
    - Within the ISR:
      - Decrement the `value` member of each Timer by 1.
      - If `value` is 0, set the `event` member to `TRUE` and reset the `value` to the appropriate countdown value. This creates an event which the main() code can use.
        - Also reset the "timeRemaining" member to the appropriate value for that timer. (Avoid magic numbers by using #defines!)
      - TimerA should trigger an event every 2 seconds, TimerB should trigger an event every 3 seconds, and TimerC should trigger an event every 5 seconds.
    - In your main():
      - Poll events from each timer by examining that timer's "event" member in a while loop.
      - If you detect an event from a timer:
        - Print an appropriate message ("A", " B", or " C")
        - Reset the "event" member to False
- **Leds_Lab06.c** library
  - Implement the functions described in Leds_Lab06.c. By this point, the process of writing functions for a library should be familiar to you.

    - Implement testing of your Leds_Lab06.c into bounce_timers.c by using the timers to make patterns of LEDs.
        - LED 1 should toggle on each TimerA event, LED2 should toggle on each TimerB event, and LED3 should toggle on each TimerC event. Use C's bitwise toggle operator as well as all three functions in Leds_Lab06.h library.
- If bounce_timers.c is working correctly, it should print something similar to the following sequence:
    - A B A C A B A B A C A B A B C A A B A C B A A B C A B A
- Check to make sure your timers are counting correctly by covering 2 of the LEDs and making sure it toggles at the appropriate frequency.

Part 2 – bounce switch:

- **bounce_switch.c**:
    - 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, appearing to bounce back and forth between the LD1 and LD8 LEDs.
    - The speed of the bounce is controlled by the switches. When all switches are up, the LED should move slowly. When all switches are down, the LED will move very quickly. The speed of the LED should change immediately when the switches are changed.
    - Use the Timer1 ISR to generate Timer events:
        - Define a struct datatype called "Timer" as in the previous part, and a module-level instance of a timer.
        - On a timer event, use the SWITCH_STATES() macro to determine the appropriate timeRemaining.
    - #define or enum{} the constants LEFT and RIGHT and use them within your while-loop and for the direction of your bouncing LED. Make a "state" variable that will hold one or the other of these constants.
    - In your main code:
        - Poll the timer event as in the previous part. If there is a timer event, decide whether you need to switch from LEFT to RIGHT (or vice versa) and shift the lit LED as appropriate.
        - Before you leave your event-handling block, clear the event flag.

Part 3 – bounce_adc:

- You will be sampling an analog voltage from the IO shield's potentiometer and displaying the resulting value on the OLED. The OLED should display both the raw ADC reading (from 0 to 1023) and a percentage (0% - 100%).

- Use the ADC1 Interrupt Service Routine to generate ADC events:
    - Define a struct datatype called “AdcResult”. The AdcResult struct should have an 8-bit member named "event" and a 16 bit member value named "voltage". Make a module-level instance of this struct.
        - The “voltage” member of this struct will serve as the center of a window used to trigger ADC events. Use a #define to set the size of this window (5 is a good size)
    - Read the 8 buffered ADC values in the ADC1BUF0 through ADC1BUF7 SFRs and average all of them.
    - If the ADC reading exits the window, generate an ADC event and update the window center.
        - The ADC reading should be able to reach 0 and 1023. You will need some special handling for these low and high cases.
    - If you have implemented the event system correctly, the ADC reading should not “flicker” or “wobble”. It should change only when the potentiometer has been moved by a human.

Part 4 – Buttons.c

- Implement the Buttons library in a Buttons.c file.
    - ButtonsInit() must set the appropriate SFR bits to set your button pins to inputs. It should change no other bits.
    - ButtonsCheckEvents() should be a fully asynchronous event checker. That is, if you tap a button once, it should only report events twice (once for DOWN, once for UP).
    - ButtonsCheckEvents() should perform “debouncing.” That is, rapid mechanical oscillations should be ignored. ButtonsCheckEvents() should only report an event AFTER it has observed a stable button reading over a number of measurements. This number is defined in Buttons.h .
        - This debouncing should occur over multiple calls to ButtonsCheckEvents(). That is, **you should not call BUTTON_STATES() more than once inside of ButtonsCheckEvents()**! You can use ctrl-f to find all calls of BUTTON_STATES() in Buttons.c. If more than one match is found, you’re doing it incorrectly.
- Within ButtonsTest.c you will:
    - Create a test harness that tests your buttons functions and ensures that they return the correct values.
    - You MUST use the buttons library. ***You should not use BUTTON_STATES() in ButtonTest.c***!

    - Unlike previous test harnesses, ButtonsTest will require human input (ie, pressing the buttons). Use a while loop to print ONLY when events are generated:
        - Here is an example of useful output (notice each event is only detected once! Also notice that simultaneous events can be detected.):

```
Welcome to mnlichte's lab6 part4 (ButtonsTest).
Please press some buttons!
EVENT:  4:---- 3:---- 2:---- 1:DOWN
EVENT:  4:---- 3:---- 2:---- 1:  UP
EVENT:  4:DOWN 3:---- 2:---- 1:----
EVENT:  4:  UP 3:---- 2:---- 1:----
EVENT:  4:DOWN 3:---- 2:DOWN 1:----
EVENT:  4:---- 3:---- 2:  UP 1:----
EVENT:  4:  UP 3:---- 2:---- 1:----
EVENT:  4:DOWN 3:---- 2:DOWN 1:----
EVENT:  4:  UP 3:---- 2:  UP 1:----
EVENT:  4:---- 3:---- 2:DOWN 1:----
EVENT:  4:---- 3:---- 2:  UP 1:----
```

    - Please note that ButtonsTest_template.c configures its timer to run significantly slower than the ones used elsewhere in this lab to make it easier to observe your code in action (ie, with debugging print statements, or by performing “simultaneous” button presses).
- Make sure that all data types, variables, macros, etc. that you code for your buttons library entirely contained within the Buttons.c file. Do not modify Buttons.h.
- Also, do not modify the Timer1 ISR in this ButtonsTest.c

Part 5 – bounce_buttons:

- Use the Buttons.c library to control bank of LEDs with the buttons.
    - You MUST use the buttons library. ***You should not use BUTTON_STATES() in bounce_buttons.c***!
- The behavior of the LEDs is dependent on the state of switches:
    - If switch 1 is down, then each Button DOWN event toggles the appropriate LEDs, and button up events do nothing.
    - If switch 1 is up, then each Button UP event toggles the appropriate LEDs, and button down events do nothing.
- Each button controls 2 LEDs:
    - BTN1 - Toggles LD1 and LD2
    - BTN2 - Toggles LD3 and LD4
    - BTN3 - Toggles LD5 and LD6
    - BTN4 - Toggles LD7 and LD8
- Your event checking code should be entirely synchronous. That is, it should only run at regular intervals. In your Timer ISR:

    - Call ButtonsCheckEvents() and store the result in a module-level variable.
- Your main() loop should be entirely asynchronous – it should only take action in response to a button event.

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.

README.md:

- Your readme should follow the 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. Reminder to use Github-flavored Markdown to improve the readability of your documentation.
    - First you should list your name & the names of colleagues who you have collaborated with.[2]
    - 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 did 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 you understand this lab or would more teaching on the concepts in this lab help?

[2] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

Extra credit:

- Combine parts 1 through 5 into a single program in bounce_ec.c with the following modifications:
    - The speed of the LED is now controlled by the pot. The LED should bounce faster as the pot is rotated clockwise.
    - Button events will "freeze" or "unfreeze" the bouncing LED.
    - If switch 1 is down, then a button down event should switch the bounce from frozen to unfrozen, or vice versa. If switch 1 is up, then a button up event should switch the bounce from unfrozen to unfrozen, or vice versa.
- You will need to initialize the timer ISR and ADC ISR. You can copy-paste the configuration code from previous sections.

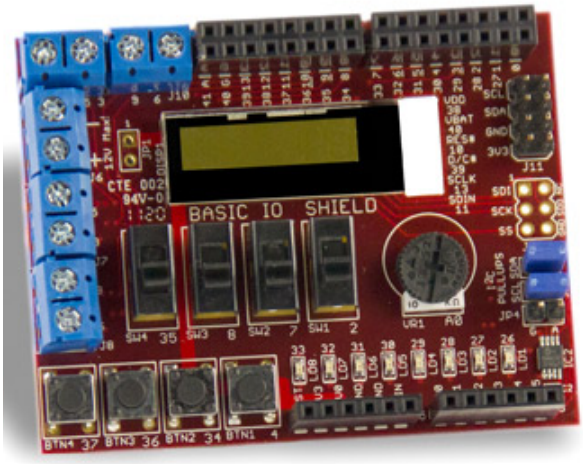
Grading

This assignment consists of 19 points. 20 points are available, so you have a 1-point "cushion".

- 3.0 - Autograded components:
    - 3 - Buttons library
        - 0.5 Buttons Init correct
        - 0.75 reads buttons correctly
        - 1.25 Event checker does not "spam"
        - 0.5 debouncing
- 16 - Human components:
    - 1 points - Part 1 – bounce_timers.c
        - 0.5 Timer messages print at correct intervals
        - 0.5 LEDs 1-3 toggle at correct intervals.
    - 3 points - Part 2 – bounce_switch.c
        - 1 – LED bounces back and forth
        - 1 – Timer ISR generates timer events properly
        - 1 – main loop is entirely reactive (no blocking code!)
    - 3 points – Part 3 – bounce_adc.c
        - 1 – OLED displays ADC value, updating continuously
        - 1 – ADC ISR generates adc events properly
        - 1 – main loop is entirely reactive (no blocking code!)
    - 2 points – Buttons.c
        - 0.5 - ButtonsInit() initializes the button SFRs correctly
        - 1 – ButtonsCheckEvents() returns events only when button state changes occur, and does not "spam"
        - 0.5 – ButtonsCheckEvents() returns events using the correct button flags, and can handle simultaneous events.
    - 3 points - Part 5 – bounce_buttons.c

        - 1 – LEDs toggle with button presses, depending on switch state
        - 1 – Timer ISR generates button events properly
        - 1 – main loop is entirely reactive (no blocking code!)
    - 2 points – code style and readability
    - 2 -- README.md
    - 1 -- combining all 3 parts into bounce_ec.c.
- Deductions:
    - If a source file doesn't compile, you'll lose all points that rely on that file
    - -1 points: at least 1 compiler warning
    - Other deductions at grader's discretion

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. It 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.

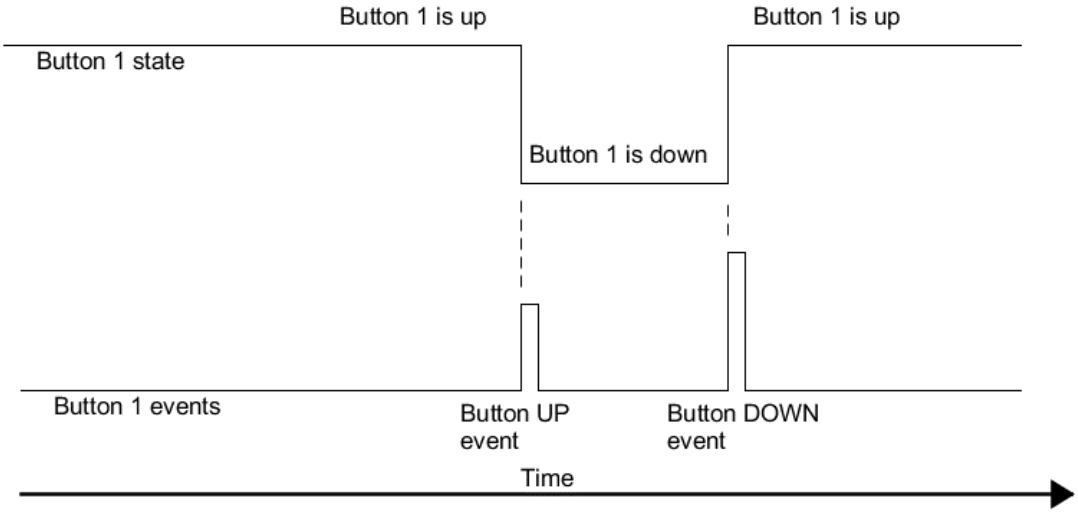
Event Driven Programming

Event-driven programming is an extremely powerful code architecture for reactive systems (things that need to respond to external stimuli). It is an essential technique in embedded code, where your device often needs to react to many events occurring quickly. This lab introduces you to the concepts of system state and event checking.

In order to understand event-driven programming, we must first introduce two key concepts: States, and events.

"State" is just a term used to refer to some temporary (but not instantaneous) property of a system. 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. Variables have a state, as well. An 8-bit number has 2^8 states, one for each possible value.

"Events" are instantaneous changes in some state. This change is driven by a change in an external state. For example, a button event checker observing that a button has switched from pressed to unpressed. The distinction is captured in this diagram:



Systems use states to determine what actions they should undertake *next*.[3] For example, when we're in the MOVING_LEFT state, the next time the LED should move it will move to the left. When a car is in the

[3] More formally, the "state" encapsulates everything you need to know about the history of any actions that got you there. The next state is a progression depends ONLY on the current state, and the event that has occurred. This is a very powerful way to decompose problems, called the Markov property.

GEAR_1 state, an upshift transitions it to GEAR_2. If an 8-bit value is in state 00000101, and it is incremented, it will change to state 00000110.

Within the examples of states above, event[4] checking has already been implied. Event checking is code that tests for changes in the triggers that the program watches: Has button S3 been pressed? Has the counter reached its limit?

In order to detect a change in a system, *an event checker must have some memory of the system's earlier state*. You cannot detect a change from a single photograph, at least two photographs from different times are necessary. Therefore, every event checker must maintain a variable that records previously observed states.

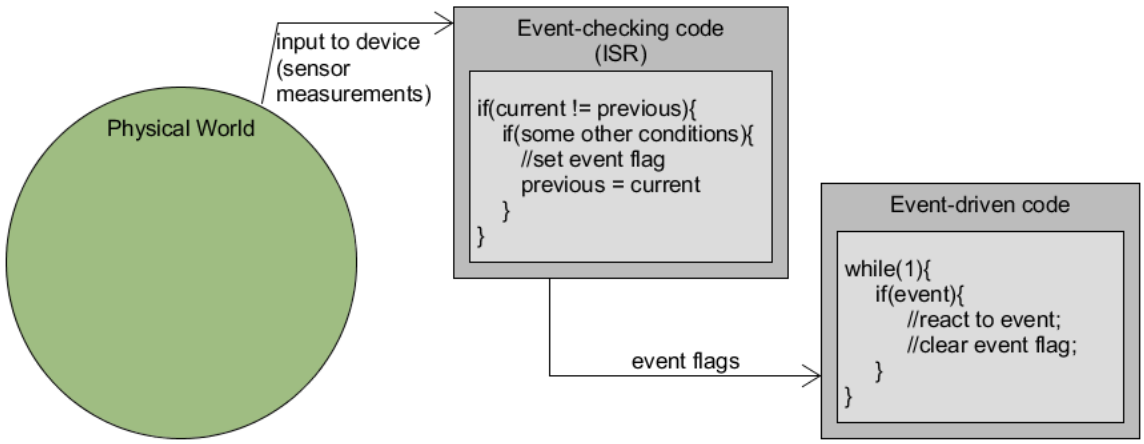
Events can also cause transitions between states elsewhere. For example, your computer monitor may react to a POWER_BUTTON_PRESSED event by switching from state ON to OFF. In this case, there are two states: The state of the button, and the state of the monitor. The event checker in the monitor watches the state of the button. When it observes a change, it reports an event to the monitor's power controller, which then switches the monitor's state[5].

This pattern will be repeated, again and again, in many different forms, for the remainder of the class.

If you want a bit more concrete view of how this looks in code (at least in this lab), this diagram illustrates the above:

[4] An event is simply some detectable change in the system. It can be a timer expiring, a voltage changing on a pin, etc. But it must be *different* than what was before to indicate a *change*.

[5] In a sense, there are 3 states in this example: A) The physical state of the button, B) the physical state of the monitor, and C) the value of the variable that the event checker uses to keep a record of A.



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 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;
```
[6]

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.

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!

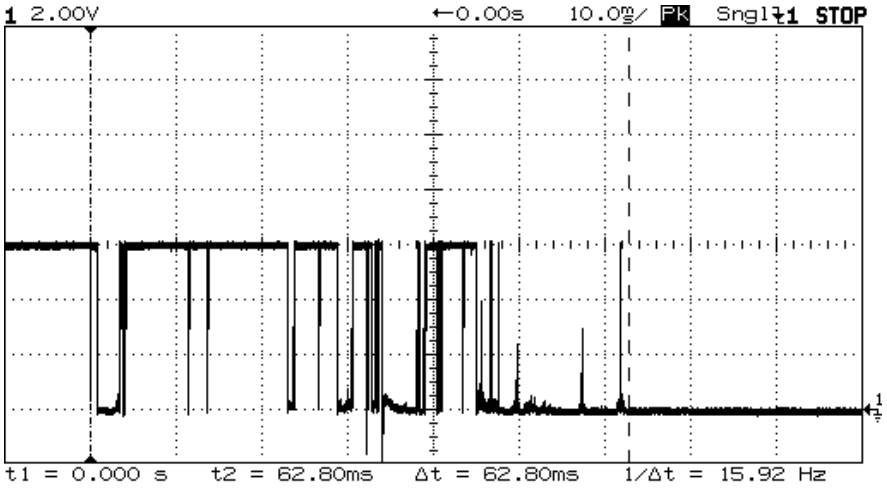
Some example test code for the Buttons library is provided below. Make sure you understand what the expected output is from the test before you code it. Tests don't help if you don't check that your program is running them properly!

```
// 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;
     }
}
```

[6] The TRISx registers are called the TRI STATE registers, and are used to set the input/output direction of each pin on port x. If the corresponding bit is set to a “0” then the pin is an output (driven by the microcontroller); if it is set to a “1” then it is an input (read in by the microcontroller).

Debouncing

The electrical signal coming into the micro from the buttons does not cleanly go from one state to the next cleanly when the button is pressed or released. Rather, there are numerous transitions between high and low over a very short period of time before the button state become stable (called switch bounce). In order to generate only a single “clean” event when the switch is pushed, or released, you will need to implement software debouncing of the buttons.



In order to effectively debounce the button, it must be sampled periodically. Simply reading it multiple times within a for loop won’t debounce the switch as the micro samples the pins much faster than the switch can bounce. There are many ways to debounce a button but for this lab, we consider the buttons to have changed state when a measurement is different from the previous state and enough time has passed. You will have to keep track of the number of calls to `ButtonCheckEvents()` for each button.

That is, the `ButtonCheckEvents()` function tracks the current button state independently of the hardware and how long it has been since the last event. As an example, if a call to `ButtonCheckEvents()`returns an up event it will not return a down event regardless of the button measurement until the debounce period is complete.

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 latch (`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 enables LEDs D1 through D5, and turns on D4:

```
// Turn on the LEDs LD1 and LD4
TRISE = 0xE0; // 1110 0000
LATE = 0x08;  // 0000 1000
```

The library you will implement in Leds.c will contain 3 separate functions 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() function 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);
```

In the next lab, you will convert Leds.h to use macros, so no Leds.c file will be needed.

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 time-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 again be triggered.

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 (a “void/void” function). 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_XXX.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()`, `OledUpdate()`, or other 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 lab, we ask you to implement timer-based events. This is a good first example of event-based programming. The ISR will handle the “ticking” of the timer. Each time the ISR runs, the time left on the timer will decrease. Once the timer hits 0, it goes off by setting an event flag. The main code continuously polls this flag in a while(1) loop. Once the main code detects that the flag has been set, it reacts by clearing the flag (in other words, “consuming” the event) and executing some response code.

In the timer ISR, as in any ISR, you must clear the interrupt flag. If this flag isn't cleared, the processor will never return from the interrupt.

In this lab the timer is configured to run at 1/8th the speed of the peripheral clock on the PIC32. 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 0xFFFF (16-bit timer). The timer we have provided 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, a bit too fast to watch it move.

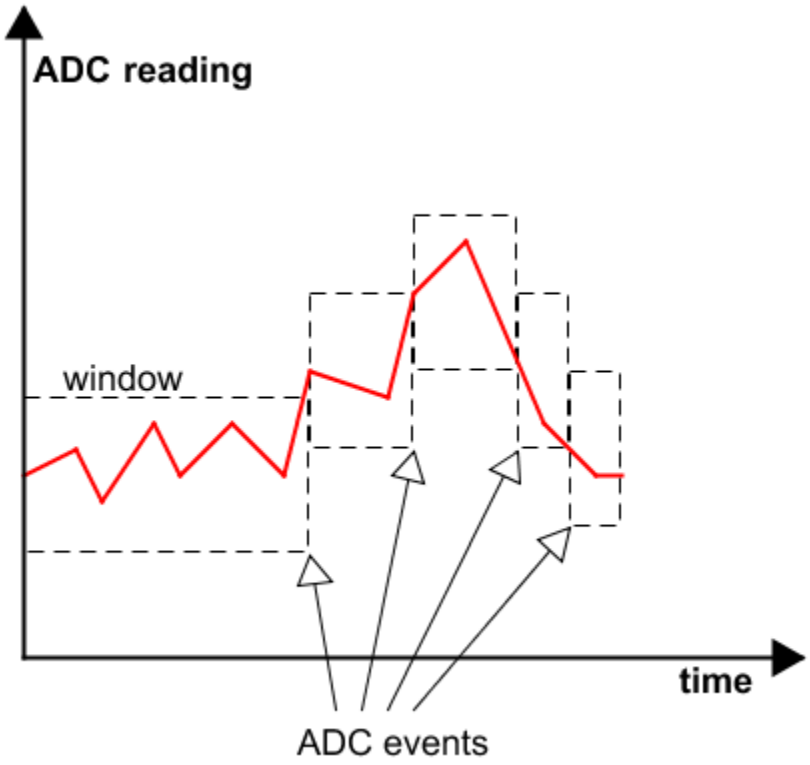
Analog to digital converters (ADC)

Think of a volume dial, an input with infinite settings between a minimum and maximum. This is what's called an analog control: it has an infinite 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 ADC Pins 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 be stuck in the interrupt, always returning and immediately reentering the interrupt.

Event after averaging the 8 buffered values, the ADC is still quite noisy, so your reading can “oscillate” between adjacent values. To prevent this from causing a rush of events, we will use a moving window to trigger ADC events. This is easily handled in code by storing the center of the window, and only moving it when the difference

between your stored center value and the measured value exceeds a threshold (3 is a good choice).



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. The units of this integer are 3.3V / 1023, or 3.23mV.
- Maintain a variable with a previously measured value. If the current value is significantly different from a previous value, then
    - update the previously stored value to your new measurement
    - set the event flag to true.

Program flow

Part 1 – bounce_timers.c:

As with the other parts of this lab, the main() code should be polling for events, while the ISR detects events. In this case, the events are all very simple timer events. A module-level variable is used to send information between the main code and the ISR. The information flow goes both ways: The ISR sends timing events to the main code, while the main code clears timer flags.

```
Main:
      initialize leds library
      initialize timers
      while true:
            if timerA event flag is set:
                  clear timerA event flag
                  print A
                  toggle LED 1
            if timerB event flag is set:
                  . . .

Timer ISR:
      Clear interrupt flag
      Decrement timerA's timeRemaining
      If TimerA's timeRemaining is 0:
            Set TimerA's event flag
            Reset TimerA's timeRemaining
      If TimerB's timeRemaining is 0:
            . . .
```

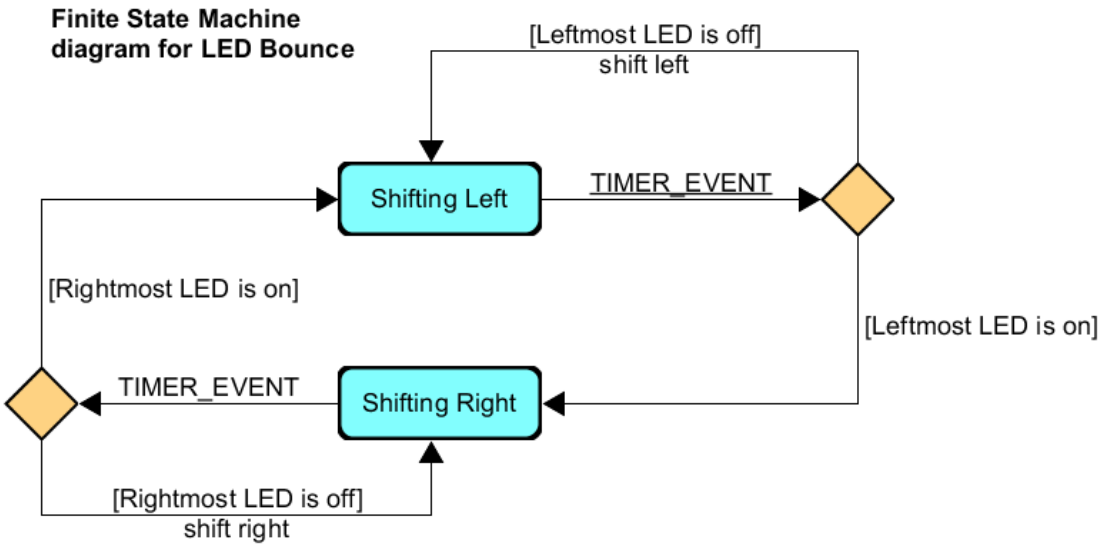
Part 2 – bounce_switch.c:

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
```

This is equivalent to the following State Machine diagram:
In this diagram, the rounded rectangles represent the state of your



main loop. When an

event is detected, your system should follow the matching transition (ie, arrow). The diamond nodes are decision nodes (not states!) and your code should pick the transition out of the diamond that whose [condition] is true.

Part 3 – bounce_adc.c:

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 event if it is different from the last sampled value. This event is then checked for in the event loop in `main()` to display the value on the OLED. 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 4 – buttons_test.c:

The Timer1 interrupt is called continuously as `main()` runs, and it will call ButtonsCheckEvents() to generate events in main. This event is then checked for in the event loop in `main()` to display the value on the OLED. It will follow the outline below:

```
while true:
```

```
if any button event flags are not clear:
      print a message indicating which events were detected
```

Part 5 – bounce_buttons.c:

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 interval, 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 light the corresponding LEDs. 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 up event:
                  toggle appropriate LEDs for that button
            else if Switch 1 is off and Button down event:
                  toggle appropriate LEDs for that button

            repeat for other buttons

            clear button event flag
```