ELEC 327 (2014) Lab #4: Simon (randomness and debouncing)

In this lab, you’ll be designing a simple variant of the game “Simon” (where one follows the patterns presented in flashing lights by pushing buttons in the right order). The code you write in this lab will directly help you with writing final code that works with your PCB.

This lab will build off concepts and code covered in labs 1 and 2. If you skirted by on those labs without really knowing what was going on or what the code you were writing was actually doing, please ask! In general, though:

- Read the Piazza post on registers and the datasheet! Hopefully it will help you to make more sense of the user guide and datasheet, and give a sense of coding with knowledge instead of randomly setting values until things work. The user guide is really long, but if you know how to navigate it, it can explain a lot of concepts very well.
- This class incorporates a good amount of coding, MSP430 specific stuff, external devices and their datasheets, specific programs (CCS, Eagle), etc. This implies a lot of googling. Using search engines to answer your questions isn’t bad. Obviously, if you’re really lost on some core concepts you need to point that out and ask, but for specifics of the above list, don’t feel bad googling things.
- Start early. You’ve probably learned this by now, but if you start the lab the day before it’s due you probably won’t finish. This means sitting down, maybe going to lab, and actually spending some quality time with it, not just reading it.

To clear up some things that seemed to be confusing a lot of people in the previous labs:

- The main(void) method that the MSP430 starts executing code in does not loop.
- Because the main method does not loop, it’s a good place to do initialization and things you only need to do once.
- You should set your MSP430 to enter low power mode at the end of your main loop, because if you do it before some code, there’s no guarantee that code is even going to execute (low power mode is literally shutting your system down, and it can’t execute code if it’s shut down).
- Page 43 of the user guide has a great conceptual guide to how you should think about entering and exiting low power modes. Read it!

Part 1: Catching button presses

Any I/O pin on the MSP430 (port 1 and 2 pins) can detect button presses. Button presses are just the MSP430 looking at the input on a pin, and doing something when the input changes from high to low or vice versa. What’s that something it does? More interrupts! This means you can be in low power mode, and then wake up to do something on a button press! The name for the interrupt vector that fires on button presses is “PORT1_VECTOR” (if you want to interrupt on buttons connected to port 1).

The syntax for the IO interrupts is very similar to that used for timers and ADC conversion.

In addition to P1DIR, P1SEL, etc, port pins also have registers like P1IE, P1IES, and P1IFG that work exactly as previous interrupt-related configuration registers. That is, changing BIT5 to be 1 or 0 on any of those registers is going to do something to how those P1.5 acts. To find out what 1’s or 0’s on those registers actually mean, read Chapter 8: Digital I/O of the user guide (it starts on page 336).
a) How would you set P1.5 to have a pull-up resistor?
b) If you set P1.5 to have a pull-up resistor and then put a button on it, where should you put the other end of the button, such that clicking the button changes the voltage P1.5 is reading?
c) Most buttons bounce, meaning if you click it once, it will oscillate between open and closed very quickly for a short time afterwards. This is bad, since these oscillations could accidentally be registered as additional button presses. If you wanted to dampen these unwanted oscillations, what could you add to your button-resistor circuit to do this? Think ELEC 241!
d) You could have all 4 buttons on one port, but notice that you only have one interrupt vector that’s called per port. How do you differentiate between button presses that occur on different pins of the same port? Related to this: how do you let the MSP430 know that you have processed an interrupt that was pending?

**Part 2: “Pre-Simon” Code**

To play Simon, you need to light up LEDs in a random order. The number doesn’t need to be perfectly random, just so much that in the game the same sequence won’t be made multiple times after a reset, and that all 4 numbers have an approximately equal chance of being selected. In class we discussed the use of a linear feedback shift register. We also discussed how the exact timing of button presses might yield usable entropy. The VLO and DCO are not synchronized, so mixing these sources might also yield useful entropy.

Create a function called `randInt()` that returns a random integer between 1 and 4; in your demonstration, you will show the result of 6 sequential calls to this function, and in a comment in your code, you will list the result of 12 sequential function calls.

Using information from questions (a) through (d), create a program called `pre_simon.c` that:

- Randomly blinks your (through-hole) RGB LED 1, 2, 3, or 4 times. Flash sequence should be detectable to the eye, and the number of blinks should be random and generated by `randInt()`.
- Gives the user some amount of time to click a button your breadboard the displayed number of times. (about 2 seconds is a good “correct period”). Button presses should not be detected during flashing.
- If the user presses the button the proper number of times, proceed to display another sequence.
- If the user fails to push the button the requisite number of times within the “correct period” or presses the button more times than specified, the code should shift into a “lose state” represented by a visual indication (examples: a different RGB LED color or rapid continuous flashing). Resetting the MSP430 should return it to normal functionality.
- Your video should have two demonstrations: (1) at least 6 successful button presses followed by one of the error types, and (2) some number of successful button presses followed by the other error type.

The first key challenge for this lab will be to properly implement debouncing in software. We discussed possible approaches in class on Tuesday, and more can be found on the web. The second
key challenge will be to implement (pseudo)random number generation on the MSP430. Your video should describe your approach for each of these challenges.

*For this assignment, we will not require a stand-alone writeup. Rather, you should submit commented source code. In your comments, label where you have answered questions (a)-(d), and list the sequence of random numbers generated by your `randInt()` function. We will peer-grade the code in class.*

**Part 3: Simon PCB**

Unlike last time, you won't be given a schematic with parts to start off with. Because of this, the assignment will be more open-ended. Available LEDs, buttons, switches, battery holders, resistors, and capacitors are listed on the course webpage. You can re-use any parts from the mood ring assignment, but the supply of RGB LEDs is limited, so you may have to supply your own. You should make sure your submitted design/PCB follows these specifications:

- Contains 4 pairs of LEDs/buttons. Lay them out in such a way that it is clear which LEDs correspond to which buttons, with the buttons easily pressed (likely on perimeter of PCB). The LEDs only need to blink, so PWM control is not necessary (but if you want to do some cool effects with it, feel free).
- Include an extra button on the RESET pin and/or a switch on the power line of the MSP430 so that the game can be reset.
- Make sure to include headers (and the sneaky resistor between VCC and RESET) so that you can program your MSP430 on the PCB (in the case of a reset button, this would be a pull-up resistor).
- Again, power the circuit with a 2032 coin cell battery.
- Make the circuit as small as you can reasonably get it, while still being able to easily access the buttons and see the LEDs.

This time, you'll benefit a lot from laying out your PCB on paper before you jump into eagle. Think of it as a board you'll actually be holding and interacting with. You might also want to print out the PCB design to try holding it.

*Again, make sure the final circuit passes the ERC and DRC, doesn’t have any unrouted parts (ratsnest command says “nothing left to do!”), and make sure to prepare it to be the proper gerber format before submitting.*