

# ECE2049 – Embedded Computing in Engineering Design

## Lab 1 – Implementing a “Simon Says” Game

---

In the late 1970s and early 1980s, one of the first and most popular electronic games was Simon by Milton Bradley. The objective of Simon was simple: the game displayed a sequence using 4 lighted buttons, each with a unique musical note. The player’s task was to repeat from memory the sequence the game had just played. When the player correctly repeated the sequence, a longer sequence was played—and at a faster rate. In this lab, you will implement a simplified version of the Simon game using our MSP430F5529 lab board.

### Lab Assignment

---

Starting with the I/O functions from the demo project, implement a Simon-like game, which displays a pseudo-random sequence using the four colored LEDs on the board the buzzer. When the player correctly repeats the sequence by pressing the corresponding buttons, a new “round” will begin by playing a longer version of the sequence. If the player makes an error, the game should indicate it by flashing the LEDs, sounding the buzzer, and writing to the LCD.

An example of a game would be as follows—in this example, the user completes four rounds before failing:

Sequence displayed by flashing LEDs	User Response (On keypad)
2	2
2, 1	2, 1
2, 1, 1	2, 1, 1
2, 1, 1, 4	2, 1, 1, 4
2, 1, 1, 4, 3	2, 1, 1, 4, <b>4</b>
“ERROR!” (Or something... accompanied by annoying flashing LEDs or sounds and such)	

### Prelab assignment (do at the start of lab)

---

When you start the lab, you should do the following:

1. **READ THE ENTIRE LAB ASSIGNMENT!** When you come to lab, you should have a good understanding of the requirements. Write down any questions you have so that you can ask them in lab.
2. To make sure you understand how the game works, play a few rounds of an online version of Simon, such as this one here: <http://freesimon.org>. (If you can’t play the game on your computer, you can find plenty of videos of the game being played on Youtube.)

Then, spend ~30 minutes doing the following tasks—this constitutes your “pre-lab” assignment:

3. **Sketch out the basic logic of your game** using a state diagram, flowchart, or by writing a basic main() function. You can find a code example for structuring your main() as a state machine on the labs page with the resources for this assignment. You do not need to implement/diagram all of the function calls

or even write code that compiles—the goal of this is to get you to think about *how* you will tackle the problem *before* you do the lab!

4. Think about how you will store the sequence of numbers that should match the user's input. How will you represent this in your code? Will each element of the sequence be represented by an integer, an ASCII character, or something else?

**When you have finished your prelab**, submit your work by doing **ONE** of the following to get a "signoff" for your prelab (which is part of your lab grade):

- (If you are working in person) Show your prelab to a member of the course staff during office hours (or ask for an appointment)
- (If you are remote) Submit your notes in a rough format (no need to be formal, so long as it's legible!) in the "Lab 1 prelab" assignment on Canvas. We will review your work asynchronously and send a response. If you have questions, feel free to include them in your submission

Your work will NOT be graded for correctness or completeness—all we ask is that you spend a bit of time *thinking about these questions, and then sync with us before you start coding*. You don't need to have everything figured out!

We encourage you to collaborate with your peers as you work on this lab—this is especially true in the design phases, like the questions here. You are encouraged to talk to your peers during lab, office hours, or any other time while you work on the lab!

**You should do your prelab the same day that you start working on the lab—we aren't setting a deadline, but it's in your best interest to do this as soon as you start.**  
**Again, your work will NOT be graded for completeness or correctness, we just want to sync with you about your design before you start!**

### Lab Requirements

---

Like the rest of your labs in this course, this lab is not a tutorial. Instead, you will apply what you have learned in lecture to expand your codebase (ie, the demo project) to complete this lab. In doing so, you will build your C programming skills and gain valuable knowledge of how embedded systems work!

How you complete this lab is up to you, but you need to meet all of the requirements listed below. In addition, here are a few things you should consider:

- Don't try to implement the entire lab at once! Try to implement small features (ie, shuffling and dealing cards, displaying a hand, etc.) by writing code for them, compiling it, and testing it, before you move onto another step. If you try to test everything at once, you will have a hard time identifying where problems occur.
- You do not need to complete the steps in the order listed below! Feel free to work on them in any order you want. For example, if you think one step is particularly challenging, you may want to try it first so that you can ensure it is correct.
- Use the debugger to help you! If you find your program not doing what you expect, think about how you can use the debugger to solve it. Can you set breakpoints to see if certain steps have been reached? Would it help to examine the state of certain variables in certain states?
- Always feel free to ask the course staff for help... that's why we're here!

## System Requirements

1. **Welcome screen:** When the game is not being played, the LCD should display “SIMON, press # to start” (or something similar). The game should revert to this welcome screen after the player loses.
2. **Countdown:** A new game starts when “#” is pressed on the keypad. The game should give a 3-2-1 countdown on the LCD before starting.
3. **Gameplay:** During each round, the game should flash a sequence on the 4 multi-colored LEDs, one LED at a time. The buzzer should sound with each LED flash. If the player correctly enters the sequence, the sequence played in the next round should be one element longer than the previous one.
4. **Some graphics:** The template for this lab demonstrates some of the graphics functions available using `glib`, the graphics library available for our board. Pick a different shape (square, circle, triangle, hexagon, decagon, etc.) to assign to each of the four buttons. When the player presses a button, the shape you selected should be displayed on the LCD with a spatial alignment corresponding to the button. That is, if you decide that the leftmost button (button 1) corresponds to a circle, you should draw a circle on the left side of the screen. If button 2 corresponds to a triangle and button 2 is pressed, then a triangle should appear more toward the middle of the screen. Similarly, if button 4 corresponds to a diamond, then when button 4 is pressed a diamond should appear on the right side of the screen.
5. **Pseudo-randomness:** The values in the sequence must be generated randomly. You can do this using the pseudorandom number generator function `rand()` in the C Standard Library (include `<stdlib.h>`). A good example for using `rand()` can be found at: [https://www.tutorialspoint.com/c\\_standard\\_library/c\\_function\\_rand.htm](https://www.tutorialspoint.com/c_standard_library/c_function_rand.htm)  
**Note:** `rand()` isn't actually random—it generates a “psdueorandom” sequence that *appears* random based on a fixed mathematical sequence. Most examples you will see online will use `srand()` to start off (or *seed*) the random number generator with a function like `time()` to get the current time. **Don't bother doing this in your lab:** using `time()` is a good idea on a general-purpose system, but not on an MSP430: on a general-purpose system, a program can ask the operating system what time it is, but we don't have an operating system to ask! Indeed, generating good randomness on an embedded system is actually a hard problem. We will build methods to tell time later, but for now we can ignore using `srand()` and have our program always use the same “random” sequence each time.
6. **Storing the sequence:** In order to check that the user's input is correct, you will need to store the sequence in memory, most likely in an array. Your game should be able to play sequences of up to length 32, at least. **Why do you need to select a maximum length for the sequence? Explain in your report.**
7. **Speedup:** The sequence should play faster as it gets longer. It is up the game designer (ie, you!) how to implement this. (Hint: Start with a single slow speed and add the variable speed last!)
8. **BONUS:** The buzzer buzzes because a pulse width modulated (PWM) signal is applied to it. Modify the `BuzzerOn()` function to play a different pitch for each LED. **Explain in your report how you achieved this.**

9. Write a high quality lab report using the instructions below. Your report should include a flow chart or other type of state diagram describing your game's functionality (you may break your game into several diagrams if that shows your functionality better).

### Example State Machine

---

It is recommended that you implement your game as a state machine. Your game will have some specific tasks (or states) that your code will perform within the main loop, such as displaying the welcome screen, showing the sequence, waiting for input, etc.

You can assign these tasks to different states (using numbers or an enumerated type) and use a switch statement to implement your game inside the main loop. An example state machine is shown below (you do not need to follow this example in your lab—it is just here to demonstrate the concept). Note how changing the variable state changes which case is executed at the beginning of each new loop.

**If you have questions on how this works, please ask the course staff and we will be happy to help!**

```
void main(void)
{
    initialize_things();
    int state;
    char key;
    state = 0;

    while(1) {
        key = KeypadGetKey();

        switch(state)
        {
            case 0: // Start screen
                // ...
                if (key == '1') {
                    state = 1; // Next iteration will go to state 1
                } else {
                    state = 0; // Otherwise, next iteration stays in state 0
                }
                break;
            case 1: // Display sequence
                // . . .
                state = 2; // Always go to the next state on the next iteration
                break;
            case 2: // Wait for input
                // ...
                break;
            case 3: // Another state... what should happen here?
                // ...
                break;
            // More states here...
        }
    }
}
```

## Coding Standards

---

This lab is designed to teach you how develop a good command of *control flow*, or using loops, if/else statements, and functions to accomplish tasks. As such, you should keep to the following *coding standard*, which will help you write code that uses good C programming practices and helps you write good control flow:

1. Avoid using lots of nested loops or if statements (> 3 nested loops, excluding the main while(1) loop, is probably bad practice)
2. Do not copypaste large blocks of code and change them slightly to implement different functionality. This is a sign you can better solve a problem with helper functions or loops!
3. Avoid using lots of global variables that modify your program's state. Instead, try to write functions that take in arguments and use return values—this will make it much easier to reason about how your code works.
4. For constant values, try to use `#define` statements to give these values names instead of having “magic numbers” scattered throughout your code. This makes it easier to understand your code when reading it later, and allows you to change these values easily.
5. Do not use `malloc()` to dynamically allocate memory—instead, all of your program's memory should be allocated by declaring variables at compile-time. As a general rule, `malloc()` should be avoided in *all* embedded systems code! On an embedded system, `malloc()` is *very* slow due to the complexity of its search algorithm, and the heap size is also usually very small.

If you find yourself wanting to write code that does not adhere points 1-3, you should rethink your control flow. Please feel free to ask us for help on this! It is very important that you learn how to write your code in a way that is easier to understand—this will make it easier for you to debug your code, and help you in future programming projects!

## Writing your Report

---

Since this lab was mainly a tutorial, the report does not need to be substantial—however, we are asking you to write one as practice for future reports. Your lab should be written in a professional style. It should be an electronically-prepared technical document like what you would submit to a fellow engineer or your boss.

Only one report is required per lab team. The report should include:

- **Introduction** (1-2 paragraphs max): Succinctly state the objectives of the lab and give an overview of what you accomplished.
- **Discussion and results:** Discuss what you did in each part of the lab and how you solved any problems. Describe what you did and **be sure to thoroughly answer and explain the questions asked in the lab assignment**. In general, this section should be as long as necessary to say what you need—no padding or fluff!
- **Summary and Conclusion** (1-2 paragraphs max): Summarize what you accomplished in the lab and what you learned. This should be a “bookend” to the introduction.
- **Appendices:** You should not need any in this lab. **DO NOT PASTE YOUR CODE INTO THE END OF THE LAB REPORT!** Instead, your code will be submitted as an archive file alongside your report, which is a lot cleaner!

**Lab reports are important. In industry, the FIRST view of YOUR work by *anybody* other than your immediate supervisor will see will probably be in WRITING!**

**Learning to be an effective communicator of technical information is probably  
THE MOST IMPORTANT  
job skill you can have.**

### Submitting your Work

---

When you are done with your report, you will submit it and your code on Canvas for grading. In order to receive a grade, you must submit **both** your code and your report online—even though you did not write much code for this lab, we will start the submission process now. Only one member of your team needs to submit files for your lab.

In addition, you must turn in your signoff sheet to the course staff—usually, you will do this when receiving your last signoff. If not, you can turn it in by placing it in the box in the ECE office, or handing it to a member of the course staff.

To submit your code for grading, you will need to create a zip file of your CCS project so that the course staff can build it. You can also use this method to create a complete backup copy of your project (perhaps to send to your partner or save for later). Unfortunately, the only reliable method for doing this is **from inside CCS using the instructions below**—do NOT attempt to just create a zip file of your code. To export your code:

1. Inside CCS, right click on your project and select "**Rename...**"
2. If you are submitting your project, enter a name in the following format:  
**ece2049e22\_lab1\_username1\_username2**, where username1 and username2 are the user names of you and your partner. (**NOTE:** Failure to follow this step will result in points deducted from your lab grade! If you don't do it, it makes a **lot** of extra work for the graders!)
3. Click **OK** and wait for CCS to rename your project.
4. Right click on your project again and select "**Export...**" then select "General" and "Archive file" from the list and click **Next**.
5. In the next window, you should see the project you want to export selected in the left pane and all of the files in your project selected in the right pane. Select all. You should not need to change which files are selected.
6. Click the "**Browse**" button, find a location to save the archive (like your R drive) and type in a file name using the EXACT SAME NAME used in Step (2).
7. Click "Finish". CCS should now create a zip file in the directory you specified.
8. Go to the Assignments page on the class Canvas website. Click on the assignment for Lab 0 and attach the archive file of your project that you just created and your report. When you are ready, hit the Submit button. Only one code and report submission is required per team.