ECE2049 – Embedded Computing in Engineering Design

Lab 2: Music Player++

Lab Assignment

The ultimate goal of this lab is to develop a system to play a tune using the buzzer and flash 4 LEDs with the notes. This lab has several objectives, including gaining further experience with digital I/O and understanding the operations of the MSP430's timing, and developing experience writing software which depends critically on the passage of time.

To provide practice with connecting external hardware, we will be adding our own buttons and LEDs. In addition, this lab has been restructured in a series of *stages*: you will build your game incrementally in different phases—connecting the hardware, using the timer, and finally building the game. The final stage of this lab is extra credit: you can extend your music playing setup to create a game (similar to Guitar Hero) where a player follows along by pressing in buttons in time with the song—to win, needs to keep you with your program!

Stage 0: Pre-lab

Similar to lab 1, you should start this lab with the pre-lab assignment described below. To start, do the following:

1. **Watch the intro video for this lab. The video has a dedicated section for this prelab!**

2. **Read the entire lab assignment**

3. Each note in the song you play is defined by two components: the pitch of the note, and the duration the note should be played. In order to play a song, you will need to find a way to store both properties for each note. You will also need to find a way to map notes to LEDs such that the same note always lights the same LED. (With only 4 LEDs and buttons, the same LED will need to correspond to more than one note—this is the case in Guitar Hero, too.)

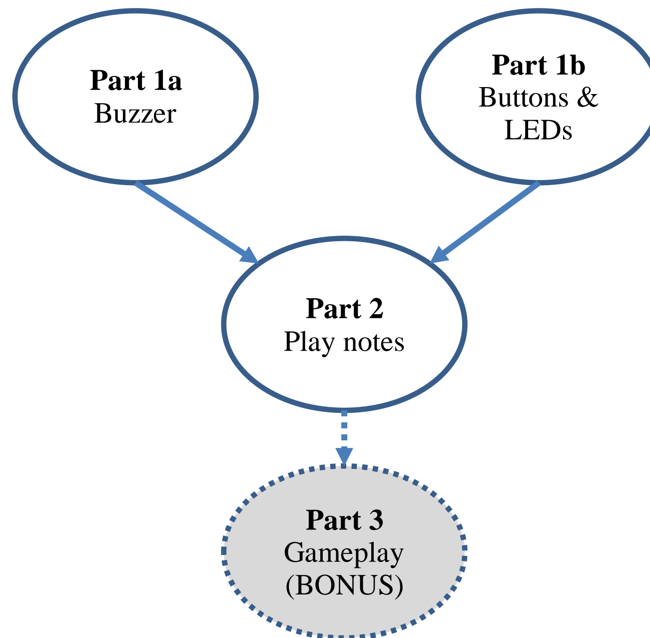
   With this in mind, think about the following: what data structure(s) will you use to store pitch, duration, and the corresponding LED for each note? **Write down your ideas.** One starting point is to use a `struct` type to store information about each note—if you want to do this, see the section Initializing Structs for more details.

4. Read Stage 2 and do step 1 (section Selecting I/O pins): select 8 available I/O pins (4 for the buttons, 4 for the LEDs) that you will use for connecting the buttons and LEDs. Write down your choices, and explain how you plan to connect/configure the pins (Input or output? Are pull up/down resistors required?). For the pre-lab phase, you do not need to write any code or draw any circuit diagrams—just write down what you are thinking. **For more info on this, see the intro video starting at 11:50.**

Write down your ideas for these questions and submit them to the pre-lab assignment on Canvas. We will review your work and provide feedback within 24 hours. You are also welcome to ask us about your work during office hours/lab sessions, and we can review your work in real-time.

Overview

This lab has several components. To simplify this process, this lab is organized into a series of *stages* where you will implement and test each component. When you finish one stage, you should submit your work on Canvas and check in with the course staff. The stages are outlined as follows:



- **Stage 1a**: Connect a small buzzer and configure it to play different frequencies
- **Stage 1b**: Connect 4 buttons and 4 LEDs to your board and write helper functions for each
- **Stage 2**: Configure TimerA2 to generate periodic interrupts and play some notes using the timer to control the duration
- **BONUS: Stage 3**: Extend your work from Stage 2 to implement MSP430 Hero: track the user's input on the buttons, compute their score, and implement all the final polish to make it a game

You can receive full credit on the lab by completing Stages 1a, 1b, and Stage 2. The final stage, which involves building the whole game, can be completed for bonus points. See the Grading Rubric for details. See the **Lab 1 Demo Video** for a demonstration of what you should be implementing for each stage.

You should start by working on Stage 1a or 1b. Doing so will require working with some extra hardware components (buttons, LEDs, a buzzer, wires, breadboard, etc.), as listed on the labs page of the course website. **If you do not yet have your extra parts**, you can start with Stage 2—read this section for details.

Also, if you are new to electrical prototyping (ie, using a breadboard), that's okay! Be sure to watch the intro video for a tutorial.

Stage 1: Buzzer

In this part, we will connect a passive, magnetic (or piezoelectric) buzzer to our launchpad. The buzzer is a simple type of speaker that can play a single tone. We will connect the buzzer to a PWM output pin (controlled by Timer B0) that will play different tones on the buzzer.

1. Using your breadboard and wires, connect the buzzer to the pin labeled **P3.5** on the MSP430 launchpad, as shown in Figure 1. You can find this pin in the bottom-right corner of the header on the left side of the board. Some buzzers are polar components (ie, they have positive and negative pins)—if this applies to your buzzer, at least one pin should be marked as such on the top or bottom of the component. The positive pin should be connected to the Launchpad, while the negative pin should be connected to ground.

Note: Remember that all circuits need a common ground: make sure your breadboard's ground is connected to a ground pin on the MSP430 launchpad.

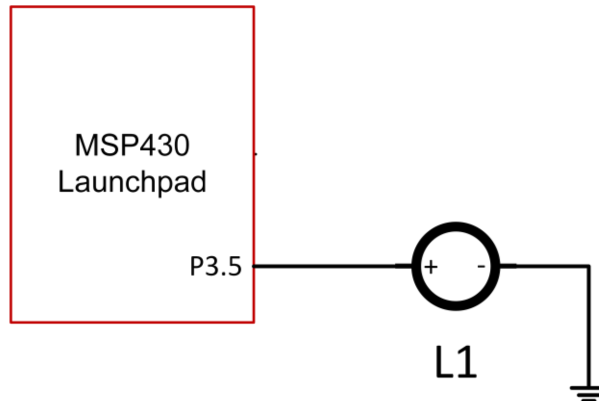


Figure 1: Buzzer wiring diagram

2. Once you have connected your buzzer, you are ready to test it. If you have not done so already, download and import the lab 2 template project from the course website. The template contains two functions, `BuzzerOn` and `BuzzerOff`, for controlling the buzzer—they are located in peripherals.c. To test your buzzer, try something like the following:

```
while(1)
{
    BuzzerOn();
    swDelay(1);
    BuzzerOff();
    swDelay(1);
}
```

If this worked, congratulations! Your buzzer works, and you should be thoroughly annoyed (mwhaha). How do you make it stop? Pause the debugger, or load a new program. If you do not hear any sound from your buzzer, check your connections and ask the course staff for assistance.

3. **Playing notes**: In order to play different notes using the buzzer, you will need to modify the `BuzzerOn` function, or create a new one, that takes in an argument to control the pitch. Examine the `BuzzerOn` function in peripherals.c, including the comments. This function uses TimerB0 to generate a PWM waveform that drives the buzzer. The period of the PWM waveform is controlled by the Timer B CCR0 register—changing the period will change the pitch of the sound. TimerB0 uses ACLK as a clock source, so the period is specified in a number of ACLK ticks.
Remember that frequency of a sound is $1/period$ and that the period--at the end of this document is a table of frequencies of an octave of musical notes in Hz. You will need to do some math to convert these notes to a number of ACLK ticks.

Note: `BuzzerOn` is defined in peripherals.c. The prototype for `BuzzerOn` is defined in peripherals.h. If you modify `BuzzerOn`, you will also need to modify its prototype. Similarly, if you create a new function in peripherals.c, you should add its prototype to peripherals.h.

4. **Test Program**: In your main() you should play a few notes to verify your buzzer works when your program starts, similar to the example in the video.
   If you have already completed stage 1b, don't remove your test code from this stage so that you can still test your buttons. In addition, light up one LED for each note that you play.

5. Once you can demonstrate that your buzzer can play different notes, you have completed this stage!

Stage 2a: Buttons and LEDs

In this part, we will connect 4 buttons and 4 LEDs to the Launchpad that will serve as our main input and outputs for the game.

Note: The instructions here are designed to supplement the intro video, which demonstrates this part in detail. **Make sure you watch the intro video!**

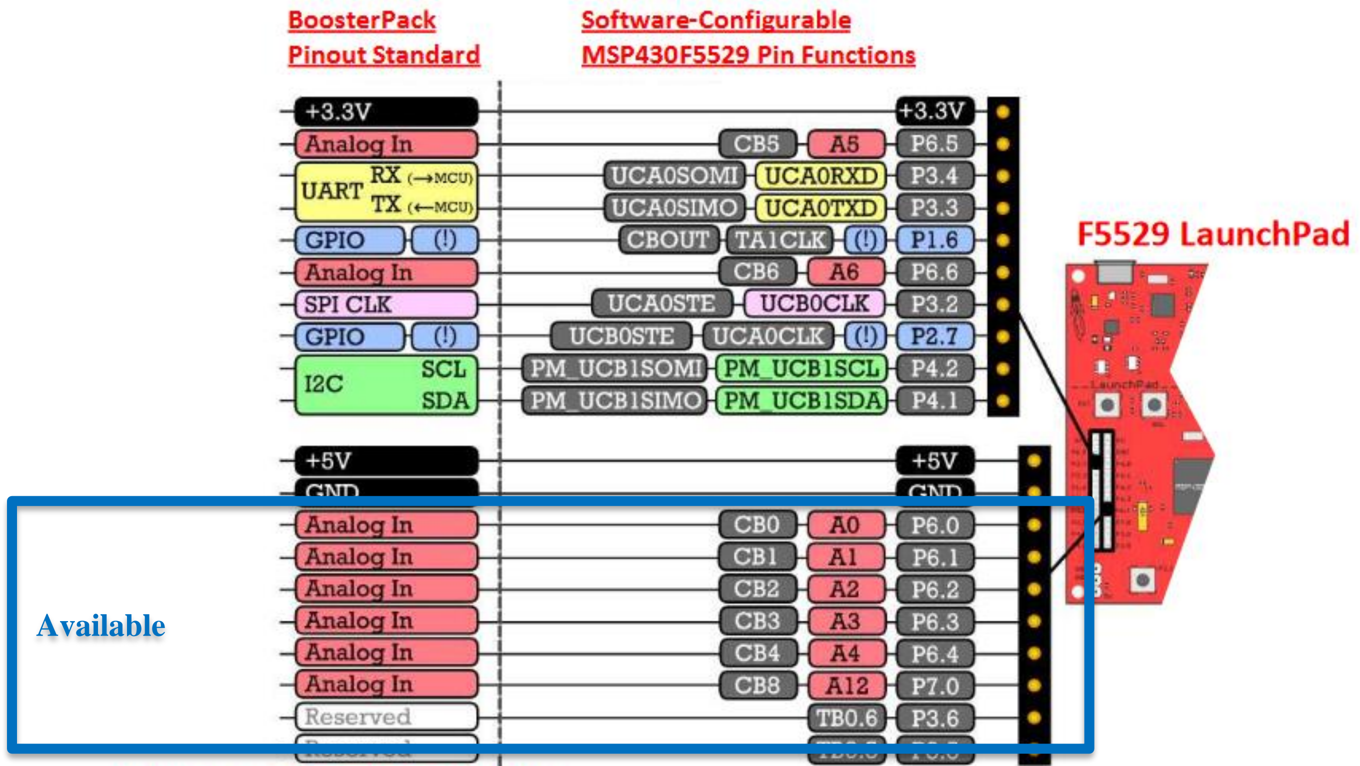
Selecting I/O pins

A key part of adding external peripherals to any microcontroller is determine how they should be connected. In this part, we will be connecting 4 buttons and 4 LEDs. These are all digital I/O devices, so we need to use a total of 8 digital I/O pins. The MSP430 launchpad has two headers that provide access to a number of pins in our MSP430. The Launchpad User's guide describes the functions available on each header, reproduced here in Figure 2. **For more information on this part, see the section of the intro video starting at 33:45.**

We need to find suitable pins to use for the buttons and LEDs. We have two requirements:

- The pin must be usable as digital I/O (ie, it has a Px.y designation)
- The pin must not already be in use for something else. We are using two other peripherals:
  - The buzzer (Stage 1a) uses P3.5
  - The LCD display module uses a number of pins, which are on the outer edges of the headers

1. Examine the header pin diagrams in Figure 2—the available pins are highlighted in blue boxes. From these, select 4 available pins to use for the buttons, and 4 pins for the LEDs. There are many possible choices, you can choose based on your preference. For example, you may wish to allocate contiguous pins (eg. P9.0-3) to the buttons or LEDs, depending on how you write the code.



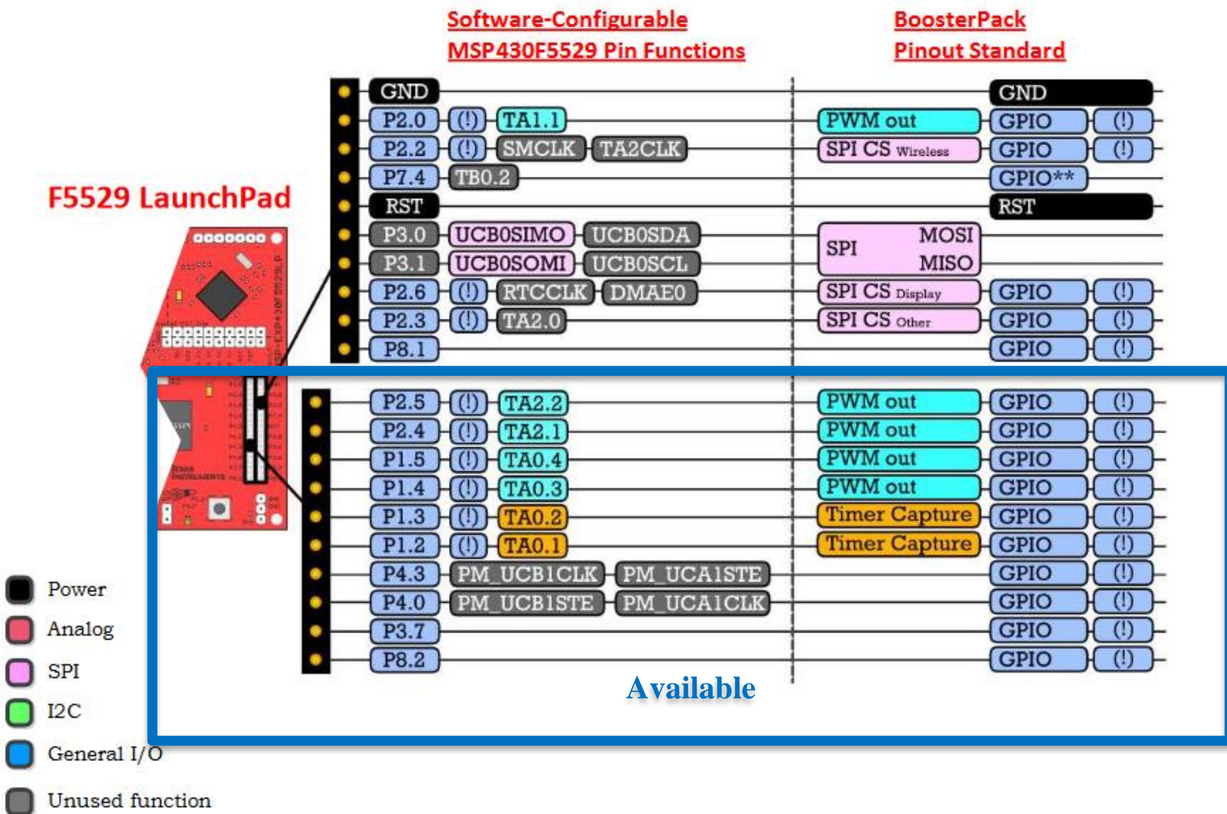


Figure 2: Launchpad Pin Reference

Connecting the LEDs

2. Using your breadboard and wires, connect 4 LEDs to the pins you selected, as shown in Figure 3. Remember that LEDs must be connected with the correct polarity—see Figure 4 (or here) for a reference. For each LED, you will need to add a current-limiting resistor: the best value will depend on the type of LED, but, as a general recommendation, a resistor between 500-1kΩ is usually sufficient. The lower the value, the brighter the LED, but using a resistor that is too small (or none at all) may burn out the LED.

   Before you connect your LED to the microcontroller, first test the circuit by connecting the wire that would go to the microcontroller (ie, the part that would connect to Px.y in Figure 3) to 3.3V—you should see the LED turn on. If it does not, check your circuit and the LED polarity, and make sure you are not using a resistor that is too large. If all else fails, try a different LED.

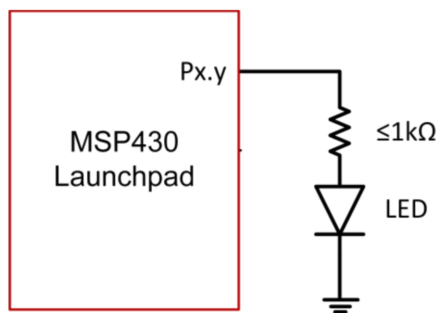


Figure 3: Example LED wiring diagram

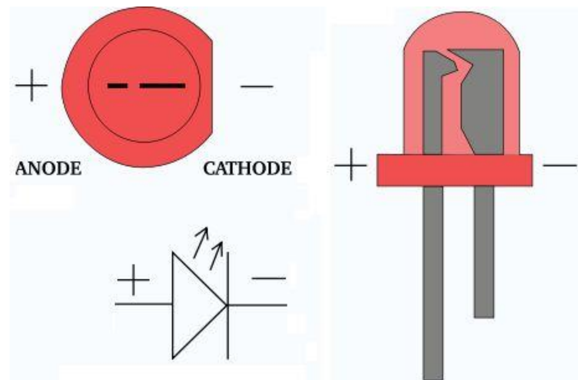


Figure 4: LED polarity diagram

3. Write a function `setExternalLeds()` to take in a variable x and set the state of the LEDs, interpreting x as a bit vector. The implementation for this is very similar to the function `setLaunchpadLeds`, as well as the examples developed in Lecture 5 (such as the decoder example). You will need to tailor these examples to use the pins you selected for the LEDs.

4. Before continuing, test your `setExternalLeds()` function by adding some code to light up each LED (you will change this once you add the buttons).

Connecting the buttons

1. Using your breadboard and wires, connect 4 buttons to the pins you selected. See the video for guidelines on working with the physical buttons.
   As we discussed in class, there are many ways to connect buttons. Figure 5 shows an example with the

simplest external wiring, which is also how the buttons on the Launchpad is connected. We recommend you follow this example, but you can do something else if you wish.

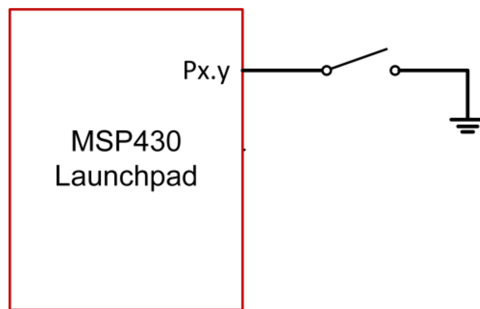


Figure 5: Example button wiring diagram

If you follow this example, you will need to configure the buttons using *internal pull-up resistors*. (Why? Explain in your report.) If you use this configuration, you will configure the buttons very similarly to the configuration for the Launchpad Launchpad buttons—you can find examples in peripherals.c (`initLaunchpadButtons()`) and in Lecture 5 (notes and the decoder example). You will need to tailor these examples to the pins you selected for the buttons. You are also welcome to use a different configuration for the buttons (eg. using external pull-up resistors, etc.).

2. Write a function to read the state of the buttons and return the state of all 4 buttons as a bit vector. The process for this is very similar to the code for `readLaunchpadButtons`, and for all the input examples developed in class (Lecture 5, the decoder example), and for HW4. You will need to tailor these examples to use the pins you selected for the buttons.

3. **Test program:** To test your buttons and LEDs, configure your main() to read in the state of the buttons and set the LEDs such that one LED controls one button, similar to the example in the video. (Do not remove any code you have already added for Stage 1a—the idea is to build up tests for each hardware component.) For example, your main loop could look something like this:

```
// . . .
while(1) {
    // . . .
    char button_state = readExternalButtons();
    setExternalLeds(button_state);
    // . . .
}
```

If you find that your buttons or LEDs do not function as expected, check your wiring and use your debugger to help identify the issue—try to identify if a) you are reading the inputs correctly, and b) you are setting the correct output pins. For example, if LED 2 is not turning on: is bit 2 of `button_state` changing when you press the button? Are you changing the appropriate output pin to turn on the LED?

If you have issues here, please do not hesitate to ask for help.

4. Once you can demonstrate your test program, you have completed this stage!

Stage 2: Play some notes

For this stage, we will play a simple melody using the buzzer, with the timing for each note controlled by our timer. **If you do not yet have your extra parts**, you can implement this stage (for now) by displaying the notes on the screen instead of using the buzzer—this will allow you to start working with your data structures and the timer.

1. **Defining your song**: Based on your ideas from the prelab, and any feedback you received, create the representation for your song by creating any data structures and defining a few notes. Your song should be at least 16 notes long, and should be composed of notes of at least four different frequencies and two different durations. See the section Defining Notes in C for details on defining notes.

   Your song **must** be stored as some type of array (or multiple arrays) that define the notes—you may not "hardcode" the song as a series of calls to `BuzzerOn`. What do you store in the array? This is up to you. As a starting point, we recommend using a `struct` to represent each note—and thus your song would be an array of structs.
   For details on working with structs, see the section

Initializing Structs for important details.

With each note, one of the four multi-colored LEDs should flash on for the duration of the note. The same note should always flash the same LED, but with only 4 LEDs, multiple notes may be mapped to a single LED.
How do decide which LED to turn on for each note? There are multiple ways to handle this—for example, you can either define or compute a mapping based on the note's frequency, or pre-define an LED as part of your song data structure.

2. We will use the timer to control how long each note is played. Configure TimerA2 to generate periodic interrupts at a rate of at least 0.005 seconds. (This resolution must be several times smaller than the duration of a single note, why?) Your timer ISR should increment a global counter, which you will use when playing the notes.

3. Before continuing, use the debugger to make sure your interrupts are working by setting a breakpoint in the ISR function. If your program reaches the breakpoint, your interrupts are firing!

4. **Test Program**: Your test program for this phase should consist of a state machine with two "states," described below. (You are not required to use a `switch`-based state machine for this, so long as your program behaves the same way.)

   a. **Hardware Test**: At startup, your program should perform the "hardware test" from phases 1a and 1b: play a few notes using the buzzer, and light up an LED for each note. Then, when a user presses one of the four external buttons, light up the corresponding LED and play a note on the buzzer (Thus, by pressing buttons, you should be able to play 4 different notes, one for each button). When the user presses one of the *launchpad* buttons, go to the next state.

   b. **Play notes**: Loop through each note in your song and play it for the amount of time specified in your data structure, lighting up an LED for each note. You **must** use the timer to keep track of the duration of each note—**no software delays!** When the song is done, go back to the starting state.

(continued on the next page)

   c. BONUS (+10): Play notes—now with user input! Since we have 4 buttons and 4 LEDs, we can map each LED with a corresponding button and have the user play along.
   To implement this part, modify your "play notes" state to do the following: While the note is playing in the "play notes" state, read from the external buttons. If the user presses the button that corresponds to the LED that is on, light up the green LED on the launchpad to show it was correct. When the song is done, go back to the starting state.

We recommend that you start the timer at the beginning of the program, and then compare the duration of each note to the elapsed time using the global count of interrupts (this is generally much easier than starting and stopping the timer).

If you have questions about how to use the timer count to play the song, or other implementation details, please do not hesitate to ask for help. We are happy to work with you on understanding this.

At this point, you should either 1) demonstrate your work to the course staff for an in-person signoff, or 2) upload a short video demonstrating your lab's functionality to the Stage 2 assignment on Canvas.

If you want to go on to stage 3 (bonus), read the next section. Otherwise, read the instructions in When you are done: Signoff and Report.

Stage 3: MSP430Hero! (BONUS)

This section is a bonus. You can stop here and receive full credit, or continue to implement the full game and receive many bonus points. See the Grading Rubric for details.

At this point, you have implemented all the major components for the game. Now, you will integrate everything to actually play the game. The major component at this stage is reading the buttons to determine the user's input while playing the song, as well as writing the actual logic for the game. This phase does not have defined steps—instead, we have outlined the major requirements for your game. How you complete them is up to you.

Before starting, think back to your work on the last lab. Is there anything you could have done differently from the beginning to make your life easier? Perhaps you used too many global variables, or you found your control flow was too complicated? Maybe you realized that naming every variable "`steve`" was a bad idea (it is). As you approach this lab, remember what you learned about software design to help you with this one!
If you have conceptual questions on how to approach specific C programming or software design concepts, please feel free to ask us—we are happy to help!

System Requirements

1. **Startup/Hardware test**: At startup, the LCD should display a welcome message. In this stage, you should incorporate your hardware tests from stage 1: play a couple of tones on the buzzer, and control the LEDs using the 4 external buttons. This state serves as a test so you can check if your hardware is working before starting the game. When the user presses S1 on the launchpad, the game should begin.

2. **Countdown**: After S1 is pressed, the game should give a 3 second countdown before playing its song. The LCD should display the time 3, then 2, then 1, then GO. While this is happening, use the LEDs to indicate some kind of countdown (count from 1-3 LEDs, etc.).

3. **Playing Notes (and User Feedback)**: To give the user feedback on whether they are pressing the notes in time, you should not play any sounds until the user presses a button. **In other words, don't just start playing the note immediately, like you did in stage 2.** If the user presses a correct button, play the corresponding note. If the user presses an incorrect note, you should call them out on this by playing an obnoxious, off-key sound.

4. **Scoring**: In Guitar Hero games, when the player falls behind, the game plays bad notes and eventually boos them off stage. As the song plays, you will need to devise a way to keep score based on the player's performance. If the player is missing notes, or not pressing notes at all, the song should stop and they should lose! **It is up to you to decide how you want to keep score and how you determine a win, loss, or failure (song ends early), but you must be able to account for both incorrect *and* missed notes.**

5. **A non-requirement**: While the user is playing the game itself, **you are NOT required to display anything on the LCD**. In other words, you are NOT required to display the typical "note highway" found in Guitar Hero games. Instead, display the current note that needs to be pressed on the four multi-

colored LEDs. Updating the LCD in time with the song is a challenging task—you may implement this for additional bonus points, but only try it after you have the rest of the lab working!

6. **Endgame**: Implement proper humiliation for losing using the display, buzzer, and LEDs. Conversely, you should implement proper player congratulations for winning. After losing or winning, the game should reset to the startup screen.

How will you make *your* game better than the rest? Multiple songs? Lighting effects? Improved buzzer routines? Player difficulty levels? Remember, as always, you only have 8KiB of RAM and 128 KiB of flash.

NASA went to the moon with less on-board computing. What can you do?

Defining Notes in C

Below is a table of an octave of notes and their frequencies. You can use these notes as components of your song—feel free to add more as well!

Note	Frequency (Hz)
A	440
B flat	466
B	494
C	523
C sharp	554
D	587
E flat	622
E	659
F	698
F sharp	740
G	784
A flat	831
A	880

Note: You may wish to create `#define` statements for your notes. This is a good idea, but DO NOT define them with single-letter names, like so:

```
#define A 440
#define C 523 // <-- Will not work!
```

It turns out that the name "C" is a macro in `msp430.h`. If you try and redefine this macro, you will receive *extremely cryptic* error messages. This is why single-letter globally-defined variable names are a really bad idea!

Instead, add some kind of prefix to your notes like this:

```
#define NOTE_A 440
#define NOTE_C 523 // OK!
```

Some example songs

- Beginning of "Three Blind Mice: E D C, E D C, F E D, F E D, …
- Intro to "Smoke on the Water": C D Eb, C D F Eb, C D Eb D C, …(repeats)

Initializing Structs

If you are defining your song using structs, note that there are several ways to initialize struct fields. Consider the example below:

```
// In C, a struct is a compound datatype, meaning a type that is
// composed of other types.  You can think of it like a container
// to store multiple pieces of information about a single "object."
// (There are no "objects" in C per se, but this is a way to create a
// similar kind of data representation.)
//
// Example:  A "thing" is composed of three "fields":
// an int (a), another int (b), and a char (c)

struct thing { // Define a struct
    int a;
    int b;
    char c;
};

struct thing s1; // Declare a struct (but don't initialize it)

struct thing s2 = {1, 2, 'X'}; // Set values for each parameter (in order)

struct thing s3 = {.a = 1, .b = 2, .c = 'X'}; // Alternate form using named parameters

// We can also use the same methods to initialize an array of structs
struct things[10] = {
    {1, 2, 'X'},
    {2, 1, 'Y'},
    // . . .
};
```

Coding Standards

The game in this lab requires some element of *real-time operation* in that it requires your code to respond to the keep track of the user's button presses within a fixed timing constraint. This constraint requires that your code to implement this part of the game be efficient—if you do too much extra work, you will notice lag in responding to the user's button presses. The goal of this is not only to teach you how to use interrupts, but to help you build your skills in writing clean, efficient code while being mindful of timing and resource constraints.

As such, all of the previous coding standards about good control flow from lab 1 still apply to this lab. In particular, try to avoid using too many nested loops or duplicating lots of code—you should be able to implement the main portion of the game with a fairly simple structure.

Note: To encourage you to use good software design, you **MUST** store the song as some type of array—you may **NOT** hard-code the notes of the song with a bunch of `if` statements! This would be a very poor design, as you would need to change the code to change the song!

In addition, writing code with interrupts has some specific coding requirements: as stated in lecture, your Interrupt Service Routines (ISRs) should be kept short since they interrupt the main execution of your program. For this reason, you should avoid doing expensive operations from inside an ISR, such as:

- Drawing any text to the display
- Updating the LCD display using `Graphics_FlushBuffer()`

- Waiting in a loop until something happens

If you find yourself wanting to perform any of these tasks in an ISR, you should rethink your program's design. Please feel free to ask us for help on this!

When you are done: Signoff and Report

Signoff

When you are finished (at either stage 2 or stage 3): you should obtain a "signoff" by either 1) demonstrate your work to the course staff for an in-person signoff, or 2) upload a short video demonstrating your lab's functionality. Please upload it to the Canvas assignment for the last stage you completed (stage 2 or stage 3).

Report and Final code submission

When you are done, write a short lab report and upload it and your code to the Lab 2 assignment on Canvas.

Your report should describe your major design choices for the components that you completed. In particular, you should at least discuss the following:

Stage 1a: Buzzer

- How did you convert note frequency (Hz) to Timer B CCR0 settings to play a note?

Stage 1b: Buttons and LEDs

- What I/O pins did you select for the buttons and LEDs? How did you configure them?

Stage 2: Playing notes

- How did you define your song data structure?
- How did you configure the timer?
- How do you use the timer count to control the note's duration?

Stage 3: MSP430 Hero (Only if you did this part)

- What are your rules for scoring?

General lab report guidelines

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.
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.

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.

Exporting your Code

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: **ece2049e21_lab2_username**, where username is your WPI username. (**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.

Grading Rubric

Points for this lab are allocated based on the following rubric:

Item	Points
Prelab assignment	10
Stage 1a: Buzzer	
Modified buzzer function to play tones	10
Stage 1b: Buttons and LEDs	
Hardware configuration and helper functions for LEDs	15
Hardware configuration and helper functions for buttons	15
Hardware test code in main() to demonstrate button and LED functionality	5
Stage 2: Playing notes	
Timer A2 properly configured with resolution set to 0.005s or less	10
Song defined using a data structure representing (at least) pitch and duration for each note	10
Playback of song with ≥ 16 notes using Timer A2 to measure note duration; Song must have notes of at least two different durations	20
BONUS: Checking buttons and lighting up launchpad LED to indicate correct note pressed	(+10)
Stage 3: MSP430Hero (BONUS)	
Playing game using LED to prompt for note, playing buzzer when correct button pressed	(+5)
Score keeping including handling for both missed and incorrect notes	(+5)
Indication of win/loss based on player score	(+5)
Report (answering questions listed in When you are done: Signoff and Report)	50
Total Points	150