ECE2049 – Embedded Computing in Engineering Design

Lab 2: MSP430 Hero!

The Guitar Hero music video games are very popular and a lot of fun. Like many popular games, they are also "conceptually simple." The game plays a familiar song while flashing lights indicating the notes the player should play on their peripheral I/O device (ie, plastic guitar). In this lab, you will use an MSP430 to implement a simple version of this game: MSP430 Hero.

Lab Assignment

Your game will play a tune using the buzzer and flash the 4 multi-colored LEDs with the notes. The player will play by entering the notes flashed using the 4 buttons on the development board. If the player can't keep up by pressing the buttons in time with the song, they lose. In doing so, you will practice using digital I/O and gain experience with writing code with precise time requirements using timers and interrupts.

This lab has several components, which are structured in a series of *stages:* you will build your game incrementally in different phases: writing some digital I/O functions, using the timer to play noes, and, finally, building the game. You have two weeks to complete this lab, but it is important that you start early so that you have time to build and test all of the components.

Stage 0: Pre-lab

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

1. **Read the entire lab assignment**

2. As part of this lab, you will need to write helper functions to control the 4 buttons on the development board, as well as the red and green LEDs (called the "User LEDs") on the Launchpad board. To start planning this process, do the following:

   - Find the 4 buttons on the development board schematic. What I/O pins do they use?

   - Find the 2 User LEDs on the Launchpad board schematic (found on PDF pages 54-57). What I/O pins do they use?

   For both, write down which pins control which button/LED, and consider how you plan to configure the pins (Input or output? Are internal pull-up or pull-down resistors required?) and write down your choices.

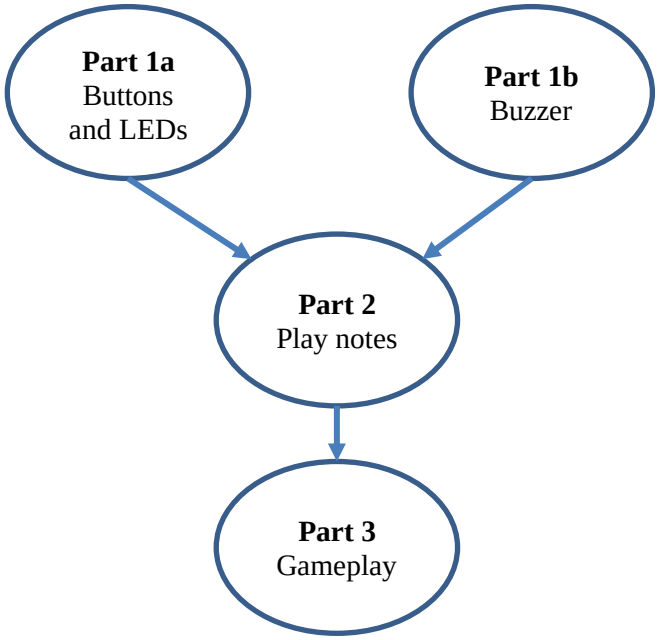
3. Each note in the song you play is defined by two components: the pitch of the note, and the duration the note should be played. In order to play a song, you will need to find a way to store both properties for each note. You will also need to find a way to map notes to LEDs such that the same note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

   What data structure(s) will you use to store pitch, duration, and the corresponding LED for each note? What length songs will you eventually want to play? Given how you choose to save your notes, how much memory will what require?

Write down your ideas for these questions and show them to the course staff during lab or office hours to discuss them. If you are working remotely, submit your responses to the pre-lab assignment on Canvas—we will review your work and provide feedback within 24 hours.

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 before building the whole lab, which is a good practice for software development. The stages are outlined as follows:



- **Stage 1a**: Write helper functions to use the 4 buttons and 2 User LEDs
- **Stage 1b**: Play different frequencies on the buzzer
- **Stage 2**: Configure TimerA2 to generate periodic interrupts and play some notes using the timer to control the duration
- **Stage 3**: 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 should start by working on Stage 1a, which involves working with Digital I/O. Later stages require using the MSP430's clocks, timers, and interrupts, which will be the subject of lectures next week. If you want to implement parts ahead of the lectures, you're welcome to view the lectures and notes from last year's version of the course to review this information early.[1]

Stage 1a: Buttons and LEDs

Your lab template already contains functions for using some of the peripherals on our development boards (ie, setLeds() and KeypadGetKey(), which are provided as a library in peripherals.c and peripherals.h. You will extend this library by implementing helper functions for two more components:

- The 4 buttons on the development board (located to the left of the LEDs)
- The two "User LEDs" on the small launchpad board, which are located on the bottom edge of the Launchpad

[1] If you do this, note that last year's version of Lab 2 was slightly different since the course was remote, but the concepts are the same.

You will extend our library code by writing helper functions to use of each of these peripherals. To do this, let's break down the required steps for each component:

User LEDs

1. Write a function (eg. `initLaunchpadLeds()`) to configure the digital I/O settings for the 4 LEDs. Based on your prelab, you should know the appropriate I/O pins and configuration settings for the LEDs—if you are unsure, feel free to check with the course staff. Your implementation for this should be very similar to the function `initLeds` in peripherals.c, as well as the examples developed in class (such as the decoder example). You will need to tailor these examples to the pins for the LEDs.

2. Write a function (eg. `setLaunchpadLeds()`) to take in a variable x and set the state for the User LEDs, interpreting x as a bit vector. The code for this is very similar to `setLeds` in peripherals.c, as well as the decoder example. Again, you will need to tailor these examples to use the pins for the User LEDs.

3. **Test program:** To test your LEDs, configure your main() to read in the state of the *launchpad* buttons and set the User LEDs using your new `setLaunchpadLeds()` function. For example, your main loop could look something like this:

```
// . . .
while(1) {
    // . . .
    char lb = readLaunchpadButtons(); // From peripherals.c
    setLaunchpadLeds(lb); // your new function
    // . . .
}
```

   If you find that your LEDs do not function as expected, use your debugger to help identify the issue—try to identify if a) you are reading the input argument correctly, and b) you are setting the appropriate bit in the output register. For example, if the green LED is not turning on: are you selecting the appropriate bit of input argument x? If you step into your setLaunchpadLeds() function, are you setting the output register correctly?
   If you have issues here, please do not hesitate to ask for help.

Development board buttons

4. Write a function, eg., `initButtons()` to configure the digital I/O settings for the 4 buttons. Note that the buttons require using *internal pull-up resistors*. (Why? Explain in your report.). Your configuration for the buttons will be very similar to the configuration for the Launchpad buttons—you can find examples in peripherals.c (`initLaunchpadButtons()`) and in the decoder example from lecture. You will need to tailor these examples to the pins for the buttons.

5. Write a function (eg. `readButtons()`) 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()` in peripherals.c, and for all the input examples developed in class (such as the decoder example), and for HW4. You will need to tailor these examples to use the pins for the buttons.

6. **Test program:** To test your buttons, configure your main() to read in the state of the buttons and set the 4 main LEDs using setLeds(). (Don't remove any code you've already added from the previous part—

the idea is to build up tests for each hardware component.). For example, your main loop could look something like this:

```
// . . .
while(1) {
    // . . . code from before ...

    char button_state = readButtons(); // Your new function
    setLeds(button_state);
    // . . .
}
```

If you find that your buttons 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 appropriate bit in the return value. For example, if LED 2 is not turning on: is bit 2 of `button_state` changing when you press the button? If you step into your `readButtons()` function, can you see a change when you read the input register?
If you have issues here, please do not hesitate to ask for help.

7. Once you have a test program that can demonstrate your new button and LED functions, you have completed this stage! Feel free to check in with the course staff to demonstrate your work.

Stage 1b: Buzzer

Our development contains a small magnetic buzzer, which is a simple type of speaker that can play a single tone. On our development board, the buzzer is connected to a PWM output pin (controlled by Timer B0) so that we can play different tones.

1. **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.

2. **Test Program**: In your main() you should play a few notes to verify your buzzer works when your program starts, and light up one LED for each note that you play. Don't remove your test code from the previous stage so that you can still test your buttons. In addition, light up one LED for each note that you play.

3. Once you can demonstrate that your buzzer can play different notes, you have completed this stage! Feel free to check in with the course staff to demonstrate your work.

Stage 2: Play some notes

For this stage, we will use our new hardware functions to play a simple melody using the buzzer, with the timing for each note controlled by our timer. This will help you test using your timer and give you practice thinking about your song and data structures.

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 Frequency Values 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 Using 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. These states will make up two "states" of your larger game.

   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.

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.

5. Once you can demonstrate playing notes, you have completed this stage! Demonstrate your work to the course staff, and continue to the next phase—building the game!

Stage 3: MSP430Hero!

At this point, you have implemented nearly 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.

However, as a first step, we recommend modifying 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.

Before building your main state machine, 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

4. **Startup/Hardware test**: At startup, the LCD should display a welcome message. In this stage, you should incorporate your hardware tests from stages 1a and 1b: 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 "#" on the keypad, the game should begin.

5. **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.).

6. **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 and light up the green launchpad LED. If the user presses an incorrect note, you should call them out on this by playing an obnoxious, off-key sound and lighting up the red launchpad LED.

7. **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.**

8. **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!

9. **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.

Bonus Points: Present your complete game to a member of the course staff for a signoff. At the end of the lab, games that go above and the requirements will receive a 15pt bonus.

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 Frequency Values 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)

Using Structs

Structs are a good way to keep track of multiple properties about a single thing, such as a note in your song. Consider the following example, which demonstrates ways to work with structs:

```
// 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'},
    // . . .
};
```

For more information about structs, you can find lots of tutorial online (such as this one), or consult your favorite C language textbook.

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!

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. 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_lab2_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.