ECE2049 – Embedded Computing in Engineering Design

Lab 1 – MSP430 Blackjack

Card games were among the first applications implemented for personal computers, and often the first applications users experience. For example, for over 20 years Microsoft Windows has shipped with small applications for a few card games (Solitaire, Free Cell, etc.). There are thousands more applications for card games where players can play against each other or against a computer—both with and without gambling—available as standalone applications or online. In this lab, you will a version of Blackjack, which is currently the most widely played casino game in the world. You will implement your game MSP430F5529 lab board, using a simplified version of the rules, in which a single player plays against the computer.

Prelab Assignment

At the beginning of the lab, you should do the following:

1. **Watch the introductory video** posted for this lab on the labs page. This should give you an overview of how the lab and prelab submission works

2. **Read the entire lab assignment.** You should ensure you have a good understanding of the requirements.

3. Sketch out the basic logic of your game using a flowchart or state diagram, 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 write code that can compile—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 will you store, shuffle, and deal cards**, and write down some ideas.

Write down your ideas in a rough format (no need to be formal, so long as it's legible!) and submit your work in the "pre-lab" assignment on Canvas. If you have any questions, feel free to add them include them and we will look them over and send a response when we review your work.
If you have already started the lab, please submit your code as-is to the prelab assignment to show that you have started—if you have any questions about your work so far or want feedback on anything specific, include them in a text file or come to office hours.

Note: Since we understand everyone will be working on the lab asynchronously, we are not assigning a specific date for the prelab—please just do it before you start the lab. The idea is to get you thinking about what you need to consider before you start writing code. Students usually find this practice helpful, so please do it! **If you are able, you are welcome to come to a live lab session or office hours, or make a private post on Piazza to discuss your prelab. If you would like to talk at a time other than the scheduled office hours, please let us know—we are more than happy to find a time that works!**

MSP430 Blackjack

Like any card game, there are many variants of Blackjack with different play and scoring rules. For this lab, we will define our own version of Blackjack specific to two players (a human player and the dealer), and with some simplified scoring rules. This section describes how the game is generally played—see the next section for specific requirements for your implementation in lab.

MSP430 blackjack is played with a standard deck of 52 cards. The object of the game is to beat the dealer in one of the following ways:

- Obtaining a hand with a score higher than the dealer, but without exceeding 21
- Forcing the dealer to draw cards until their hand exceeds 21

A typical game structure proceeds as follows:

1. To start the game, the player is dealt two cards into their hand. The dealer draws one card, which is visible to the player.

2. Each player's hand has a "score" based on the value of their cards: cards 2 through 10 have a value equal to the value on the card, face cards (Jack, Queen, King) are all worth 10 points. Aces can be worth either 1 or 11 points, whichever results in the best hand (more on this later).

3. After the cards are dealt, it is the human player's turn. The player has two options:
   - "Hit": Request another card to try and improve their hand
   - "Stay": Stay with the cards dealt and allow the dealer to play

   The human player can continue requesting cards until they decide to stay. The player's goal is to try and make their hand as close to 21 without going over, in order to obtain a hand with the best chance of beating the dealer. If the player's score exceeds 21, they "go bust", meaning they lose and the game ends.

5. If the human player holds and has a score < 21, it is the dealer's turn. The dealer then has the same options as the player. As in most casino games, the dealer will draw more cards until their score reaches a preset value, such as 17, before staying. The dealer can also "go bust," just like the human player—in this case, the human player wins.

6. After the dealer has stayed, a winner is determined based on the final score of each hand. The winner is the player with the highest score that does not exceed 21.

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

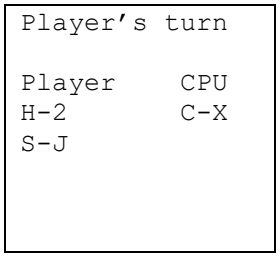
1. **Welcome screen**: When the game is not being played, the LCD should display "MSP430 Blackjack" and "Press S1 to start" (or something similar), prompting the user to press one of the Launchpad buttons to start the game. After a game ends, your game should return to the welcome screen and allow the player to play again. When in the welcome screen, a new game starts when a button is pressed.

2. **Shuffling**: As in any casino game, the cards used must be shuffled. You can shuffle the cards using the pseudorandom number generator function `rand()` in the C Standard Library (include <stdlib.h>). When starting a game, you should prompt the user to "cut" the deck: print a single digit to the display and have it "loop" from 0-9 until the user presses a button. When the button is pressed, use this value as the "seed" for the random number generator using the standard library function srand() and the number entered as the "cut"—you can either manipulate this value in some way, or use it directly.
   You can find documentation for rand() and srand() online, or in CCS help.

3. **Dealing cards**: When shuffling and dealing cards, remember that you must keep to a single, standard deck of 52 cards. Thus, **you must be sure that each card can be dealt only once per game.** How will you store the cards dealt to each player? How will you shuffle the cards and ensure they are only dealt once?

   There are many valid ways to do this—how you implement it is up to you. For example, you could create an array of shuffled cards at the start of the game and draw new cards from the beginning. Or, you could pick a new card each time you need one, and check to make sure it has not been drawn before. Whatever you decide, describe your solutions to these problems in your report.

4. **Displaying cards**: While the game is being played, you must display the cards dealt to each player (that is, the human player and the dealer) on the LCD. An example way to do this is shown below:



   Here, each card has the format A-B, where A is the "suit" of the card (Hearts, Clubs, Diamonds, Spades), and B is the "rank" (Ace, numbers 2-10, Jack, Queen, King). To simplify matters, you can represent each suit and rank by a single character (eg. use 'X' for 10, 'Q' for Queen, etc.). In the example above,

the player is holding the 2 of Hearts and the Jack of Spades, and the dealer has the 10 of Clubs.

Implementing this step requires you to build strings to display on the screen—see a note in the next section for more information on how to do this. Also, you are welcome to implement something more elaborate, if you prefer. The graphics library provides functions for drawing lines and shapes, if you wish to use them.

5. **Decisions:** On the player's turn, the player can request another card by pressing S1, or can stay by pressing S2. The player can request additional cards multiple times. If you like, you can set a maximum on the number of cards that can be in a single hand (for example, to prevent drawing cards off the screen).

6. **Turn Indicators**: To indicate when the player needs to provide input, light up the green LED when it is the user's turn to play, and light up the red LED when the delayer is playing.

7. **Scoring**: To compute the score for each player's hand, use the rules defined in the previous section for the point value of each card.
   Remember that an ace may have a value of 1 or 11, whichever results in the better hand: this means that an ace should count for 11 provided it does not make the hand bust—if it does, the ace should count for 1 instead. To simplify matters, you can start building your game by always counting an ace as 1 point, and then adding this feature later. Explain how you compute scores in your report.

8. **Endgame**: When the game ends, you game should indicate which player (human or dealer) won. Using the LEDs and LCD, you should implement proper celebration if the player wins, or humiliation if the player loses. Be creative!

9. **Edge cases**: If you find any conditions not well-defined by the specification (eg. how to handle certain scores, ties, etc.), feel free to ask the course staff for clarification. Ultimately, how you handle these cases is up to you, so long as you document them in your report.

10. **BONUS**: Can you think of any extra fun features you'd like to implement? Perhaps you would like to display the player and CPU scores on the screen, a fancy display for cards, or something else? If you implement extra features, you may be awarded with extra credit—discuss your idea with the course staff first for details.

11. 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). All flow charts should be computer generated.

(Continued on the next page)

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, shuffling and dealing cards, 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, button_state;
      state = 0;

      while(1) {
            button_state = readLaunchpadButtons();

            switch(state)
            {
                  case 0: // Start screen
                        // ...
                        if (button_pressed == 0x1) {
                            state = 1;  // Next iteration will go to state 1
                        } else {
                            state = 0;  // Otherwise, next iteration stays in state 0
                        }
                        break;
                  case 1: // Shuffle and deal
                        // . . .
                        state = 2; // Always go to the next state on the next iteration
                        break;
                  case 2:  // Display
                        // ...
                        break;
                  case 3:  // Another state... what should happen here?
                        // ...
                        break;
                  // More states here...
            }
      }
}
```

Building Strings

A challenge you will face in this lab and many others is building strings from various data (in our case, the value of a card) to display on the LCD. This can be a challenging task, and we will discuss various methods to do this throughout the term depending on the type of data you want to display. For displaying card values, a good way to start is to build a helper function like this:

```
void display_card(char suit, char rank, int x, int y)
{
    unsigned char str[4];  // Declare a buffer of 4 chars, which we will fill
    str[0] = suits[suit];
    str[1] = '-';
    str[2] = ranks[rank];
    str[3] = '\0';

    Graphics_drawStringCentered(&g_sContext, str,  AUTO_STRING_LENGTH,
                                x, y, OPAQUE_TEXT);
}
```

If you have taken a C programming course, you may remember that the C standard library has many functions for formatting strings like sprintf(). These functions consume a lot of memory, stack space, and are very slow, **so we discourage their use in this class**. Instead, you should build strings similar to the methods above. Please ask the course staff if you have questions on this.

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: **ece2049e20_lab1_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.