EGR 227: Final Project

Name: Tegran Grigorian

Date: December 6, 2025

Table of Contents

Objective:

We were tasked with creating a target game with a microcontroller. This target game is similar to a carnival where we need to hit LED's on time. This program successfully completes all requirements of this project. **Timer0** is used for the game time of 30 seconds, an interrupt is then used to count the elapsed time. Two PWMs were used, one PWM used **Timer4 and CCPR2**, this was the servo. The other PWM was **Timer2 and CCP1** which controlled the Piezo Speaker. All other objectives of the project were met including a functional game and successful demonstration.

Besides the required tasks, three extra requirements were added: GUI menu, test section and Simon says. The GUI menu is another state machine that allows the user to move around the program's interface and interact with different sections. Furthermore, the GUI added a back button in the form of the "#' button on the keypad. When the "#" button is pressed in a state where user inputs can be captured, it brings the user back to the main menu.

The test section is a testing workflow derived from old commented out code that ensures all hardware components are operational. Lastly, Simon says is a separate game that employs the functions and hardware from the target game. Simon says requires the GUI to be present for the user to easily change between target games, Simon says and other sections like tests. Simon says and its rules are more discussed in the procedure.

The procedure is a chronological construction of the project, it is very long and detailed. Meanwhile, the result is a compressed holistic definition of the project. The final state diagram and project structure is defined in the results while each component and their development is described in the procedure.

Procedure:

Hardware Development

First, I created a schematic for the microcontroller. This schematic needed to include the PIC microcontroller, speaker, motor, LEDs, LCD, button, and a keypad. The PWM pins chosen were RB1 for the speaker and RD7 for the servo.

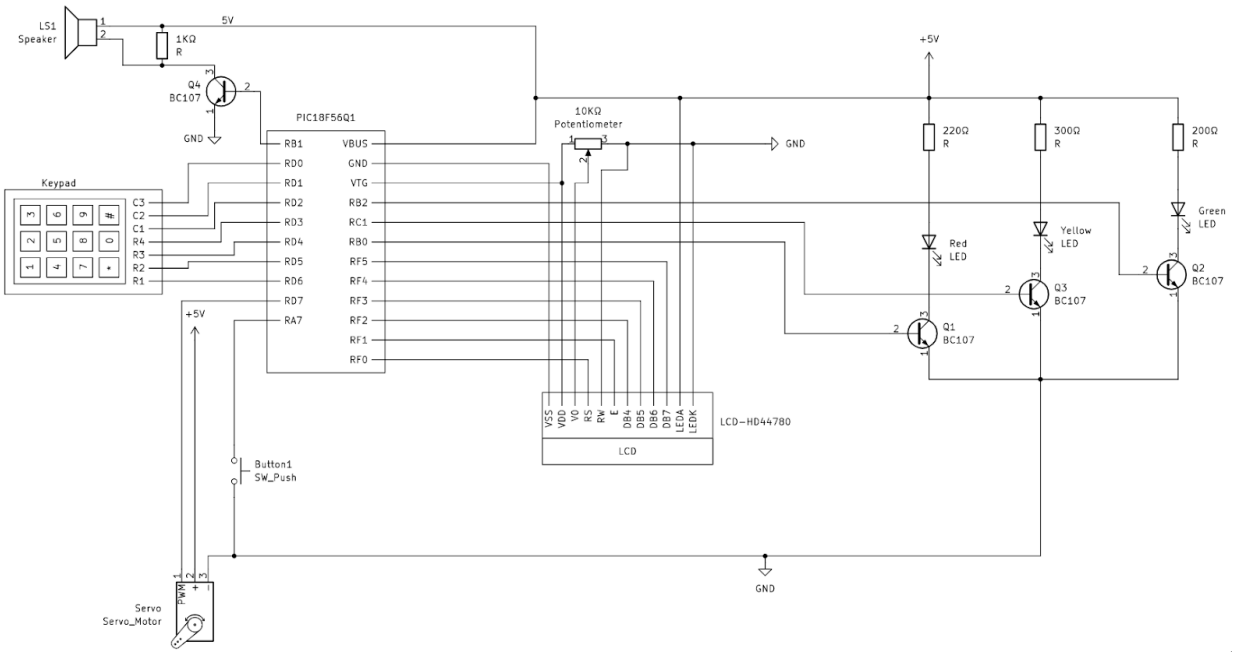

Figure 1: Schematic of Microcontroller

After creating the schematic a Solidworks model for the arcade box was created. I will not show each individual component, instead a finished assembly then what the model looks like in real life. Both of these figures are below.

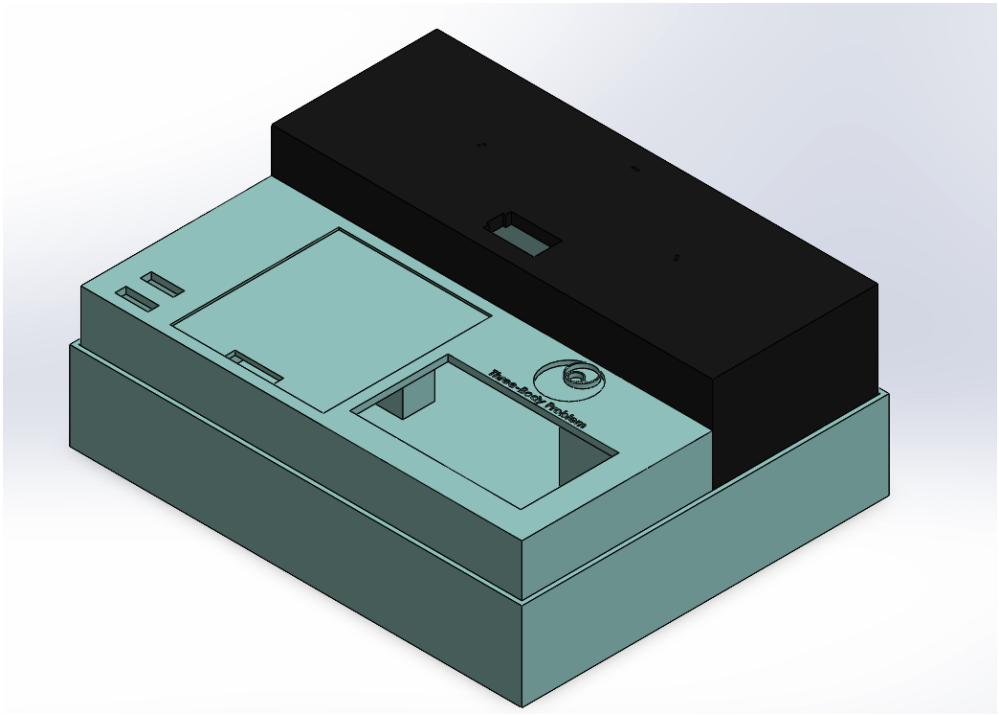

Figure 2.) Solidworks Assembly For Arcade Box

Figure 3.) Real Life Comparison

Code Creation

Once the schematic was completed, I began creating the code. First, I created a hardware file so that all the pins were in one central location. This centralized pin management in hardware.h prevented errors from scattered port definitions. Next, code from previous labs were added to ensure the hardware was working.

During this process, I discovered that the LED wiring was incorrect—the MOSFET was on the wrong side. After fixing the circuit and schematic, the LEDs were working properly. By employing old code to validate my hardware, I avoided many issues. After moving all the old code in and testing all the components, I began coding the game.

Timer0 is used as a periodic clock source for the game system. To maintain smooth and consistent timing, Timer0 is configured to generate interrupts every 10 ms. The short period allows a fast "refresh rate" of the game to ensure nothing appears laggy. Our timer will tick at FOSC / 4, which is 1 MHz. We are using a 8 pre scalar, this moves us to 8μs. To get the 8μs to 10 MS, we need to multiply it by a preload of 1250. Finally, this mathematical expression below will give us the time per interval which is our desired time of 10 ms.

$$T_{int} = \frac{1250 \times 8}{1\ \text{MHz}} = 10\ \text{ms} \quad (1)$$

Next, in order to achieve a one second time, we need to count how many times the interrupt fires. We can call this iteration 100 times in order to account for one second. The math for this is below.

$$n = \frac{1s}{10 * 10^{-3}s} = 100\ \text{ticks for 1 second} \quad (2)$$

Then simply, with 1 second, we just need to multiply that by 30 to get a proper elapsed time for the game timer. Obviously, I will not show that math, it's 1st grade math.

Finally to get the preload and TMR0H and TMR0L, we can do this mathematical expression.

$$(0\text{xFFFF} - 1250) = 0\text{xFB1E} \Rightarrow \text{TMR0H} = 0\text{xFB},\ \text{TMR0L} = 0\text{x1E} \quad (3)$$

In software, the ISR increments a tick counter, and every 100 interrupts represents 1 elapsed second, allowing accurate 30-second game timing. The code that initializes Timer0 for this operation is shown below.

```
void game_timer_init(void) {
    T0CON0 = 0x00; // timer setup
    T0CON1 = 0x00;
    T0CON1 |= (4 << 4); // fosc / 4
    T0CON1 |= (3 << 0); // 1:8
    T0CON0 |= (1 << 4); // 16 bit mode

    // preload for 10ms at Fosc/4 with prescaler 1:8
    unsigned short preload = 0xFFFF - 1249;
    TMR0L = (unsigned char)(0x00FF & preload);
    TMR0H = (unsigned char)((0xFF00 & preload) >> 8);

    // initialize variables
    elapsed_seconds = 0;
    is_running = 0;
    timer_flag = 0;

    // enable interrupt
    PIR3 &= ~(1 << 7);
    PIE3 |= (1 << 7);
    IPR3 &= ~(1 << 7);
    INTCON0 |= 0xE0;
}
```

Figure 4.) Timer/Game Timer Initialization

To wrap off this timer initialization and timer math, I'll show the interrupt code which implements the logic derived from the mathematical expressions above.

```
void __interrupt(irq(TMR0)) Timer0_GameISR(void) { // interupt function
    static uint8_t tick_count = 0; // avoid overwritting variable of same names
    if (PIR3 & (1 << 7)) {
```

```
            PIR3 &= ~(1 << 7);
            // preload for 10ms at Fosc/4 with prescaler 1:8
            unsigned short preload = 0xFFFF - 1249;
            TMR0L = (unsigned char)(0x00FF & preload);
            TMR0H = (unsigned char)((0xFF00 & preload) >> 8);

            if (is_running) { // if timer is running, increment tick count
                tick_count++;
                if (tick_count >= 100) { /// our calculated tick count
                    tick_count = 0;
                    if (elapsed_seconds < 30) { // keep counting up to 30
                        elapsed_seconds++;
                    }
                    if (elapsed_seconds >= 30) { // stop timer at 30 seconds
                        is_running = 0;
                        T0CON0 &= ~(1 << 7);
                    }
                }
            }
        }
    }
```

Figure 5.) Interrupt Code Derived From Our Mathematical Expressions

However, I had a lot of issues with the game timer. The game timer employs Timer 0, and was not counting down properly. To debug this, I had the onboard LED flash on every clock overflow. What I noticed is that as soon as the game started, the onboard LED went on and never turned off. This indicated that the clock was constantly getting overflowed. After digging through the code, I noticed my LCD code used a delay propagated by Timer 0. To fix the issue, I instead used Timer 1 for the LCD delay and exclusively kept Timer 0 for the game. Finally, the time was counted down properly from 30 seconds to 0 in 30 real-time seconds.

The `update_lcd_game_screen` function uses sprintf to format the two-line LCD display with dynamic hit/miss counts and remaining time. Line 1 displays "HITS:XX TIME:XX" and Line 2 displays "MISSES:XX". This function is called every frame during `STATE_PLAYING` to continuously update the display, and also called in `STATE_GAME_OVER` to show the final score with variable-width format specifiers ensuring left-aligned display.

Next, I began coding the LEDs. The LEDs have 3 different modes: `GREEN_RED_YELLOW`, `YELLOW_RED_GREEN`, and `RANDOM_STATIC`. Both `GREEN_RED_YELLOW` and `YELLOW_RED_GREEN` last for 1-second intervals then restart at their beginning. Meanwhile, `RANDOM_STATIC` remains on until the player hits it.

```c
typedef enum {
    MODE_GREEN_RED_YELLOW, // LEDs light from Green->Red->Yellow
    MODE_YELLOW_RED_GREEN, // LEDs light from Yellow->Red->Green
    MODE_RANDOM_STATIC // One random LED
} LEDMode;
```

Figure 6.) Led Mode Code

After downloading this onto the microcontroller, I had an issue with the `update_led_targets` function. This function defines the LED modes and their intervals by using the game timer. I had an issue with capturing if a second passed. Initially, I used an iterator that would count up with the timer, incremented via an interrupt. This would then tell the LED that a second had passed. However, this solution became verbose and led to a race condition between the state and led functions.

To solve this, I gave full write control to the state and timer functions, then created a function in the timer to get the current time in seconds. If the current time does not equal the last second, then one second has passed and we need to move to the next LED. Each LED sequence has an array that contains what LED should be lit. Finally, the LED sequence was properly moving in the right direction and time interval.

The random selection was very simple. A random number would be generated then modded with 3 to be either 0, 1, or 2. Then this is matched with a LED mode. The seed is 42, the answer to the universe.

```c
void select_random_led_mode(GameState *game) {
    game->led_mode = rand() % 3;
    game->led_sequence_step = 0; // reset led sequence step
    game->current_led = 3;
}
```

Figure 7.) Code for Random LED Mode

Next, I needed to make a test to see if the keypad was being properly handled and read. I first commented out my main code that currently runs the game. Then I added code that included the keypad.h file and had an if condition for keys 4 and 6. Then when those conditions were true I would print to the screen. This approach allowed me to debug any issues with the keypad.

Unfortunately, an issue was discovered—thankfully, it wasn't too complicated. I wired my

rows and columns backward somehow. After fixing that, it worked and the LCD was showing the prints I desired for each input. After this, I cleaned up the keypad file and created a structure for the important keypad information.

```
typedef struct { // keypad structure
    uint8_t cols;
    uint8_t rows;
    uint8_t pressed;
} keypad;
```

Figure 8.) Keypad Structure Code

I still kept the keypad code in main and added onto it calls to the servo. The servo PWM uses **Timer 4 and CCPR2**. Thankfully, the servo worked on the first try and I didn't have to mess with the code that I used from lab 9. I added new limits to the servo for ±60 degrees instead of 90, then I stored all possible servo states in an array. After I finished the keypad and servo integration, I commented out the test code and added back the main code. The function integrating the keypad to the servo was then added to the state file as `handle_servo_movement`.

Currently, I was able to move the servo, observe the LEDs ticking properly, and had a functional timer. One last major addition of button and score logic was needed. The button logic was simple to add—a simple ternary operator managed the output. Next, I added the `handle_fire_button` function which would first check if it was pressed, then it would see if it was a hit or not. The hit logic was implemented in another function called check_hit. If you hit, a sound effect would play, LEDs would turn off, the LED timer would be reset, a new mode was selected, and a new sequence was added.

```
if (check_hit(game)) { // do we have a hit or miss
    game->hits++; // increment hits
    play_hit_tone(); // play hit sound
    all_leds_off(); // turn off leds
    game->current_led = 3; // reset current led
    select_random_led_mode(game); // select new led mode
    game->led_timer = 0; // reset led timer
    game->led_sequence_step = 0; // reset led sequence step
}
```

If you miss, the miss sound would play and the miss counter would increase.

```
else { // you missed
    game->misses++; // increment misses
    play_miss_tone(); // play miss sound
```

```
}
```

Figure 9.) Handle Fire Button Code

Finally, the LCD, game state, and previous button state were updated with the new values and the game would continue. The `check_hit` function is very simple—it checks the servo position and assigns an aimed LED, then it checks if the current LED equals the aimed LED and is not the default value. Then if the current equals the aim, you hit; if it's not, you miss.

Now we are on the worst part—the stupid speaker. Initially I had the speaker wired to RC0, obviously that didn't work. I noticed this after trying to add a PWM to RC0. Instead, I changed it to RB1 and looked at older PWM code to use the proper initializations. The speaker uses **Timer 2 and CCP1**, which is different from the servo PWM and other functions. The hit and miss functions are just simply a difference of duty cycle and volume. The speaker, although obnoxious, didn't take too long to get working.

System Structure and State Management

Finally, the target game is finished. The structure goes as follows: the state.c holds the functions and structure to operate the game and handle the components. Each component has its own file and contains its own functions or structures. These are then communicated to the state file. The state file is then communicated by the main file. The main file is a simple state machine that moves the state structure between the following states.

```
typedef enum {
    STATE_INIT,        // initial target game state
    STATE_IDLE,        // waiting for fire button
    STATE_PLAYING,     // game in progress
    STATE_GAME_OVER,   // done
} SystemState;
```

Figure 10.) Structure For The System States In Target Game

State Handler Implementation

The game state is managed through the GameState structure and four primary state handlers. Each handler manages specific game phases and transitions. The `init_game_state` function initializes all game variables at startup, setting the servo to center position (`POS_3`), LEDs to the initial mode (`MODE_GREEN_RED_YELLOW`), and zeroing the hit/miss counters.

STATE_INIT Handler:

When the player selects the target game, the initialization handler displays a 3-second splash screen. During this time, the servo is positioned at center, the debug LED flashes every 125ms (8 Hz), and the speaker is silenced. The handler uses a local static timer variable to track elapsed time, incrementing every main loop iteration. Once 3000ms have passed, the screen transitions to display the initial score ("HITS:0 TIME:30") and moves to `STATE_IDLE`.

```
void state_init_handler(GameState *game) {
    static uint32_t init_timer = 0;
    static char screen_written = 0;

    if (!screen_written) {
        write_command(0x01);
        __delay_ms(10);
        write_string("  TARGET  FUN  ", 1);
        write_string("   EGR 227     ", 2);
        write_string("Tegran Grigorian", 4);
        screen_written = 1;
        servo_set_duty_cycle(servo_position_to_duty_cycle(POS_3));
    }

    init_timer++;
    if (init_timer % 125 == 0) {
        LATC ^= 0x80;  // toggle debug LED at 8 Hz
    }

    if (init_timer >= 3000) {
        LATC &= ~0x80;
        write_command(0x01);
        __delay_ms(10);
        write_string("HITS:0  TIME:30", 1);
        write_string("MISSES:0       ", 2);
        game->current_state = STATE_IDLE;
        screen_written = 0;
        init_timer = 0;
    }
}
```

Figure 11.) STATE_INIT Code

Figure 12.) STATE_INIT LCD Screen(remember my third line is broken, please don't mark me)

STATE_IDLE Handler:

In the idle state, the game waits for the player to press the fire button. The servo remains centered at POS_3. When the fire button is pressed (detected via rising edge using the previous button state), the handler calls `reset_game_state` to clear counters, then starts Timer 0 for game timing and resets the LED timer tracker. The state transitions to `STATE_PLAYING` with the onboard debug LED enabled.

```
void state_idle_handler(GameState *game) {
    static char servo_centered = 0;
    keypad key = numpad_press();
    if (key.pressed) {
        uint8_t key_num = numpad_value(key.cols, key.rows);
        if (key_num == 12) {
            all_leds_off();
            game->current_state = STATE_MENU;
            return;
        }
    }

    if (!servo_centered) {
        game->servo_pos = POS_3;
        servo_set_duty_cycle(servo_position_to_duty_cycle(POS_3));
        servo_centered = 1;
    }
```

```
    char fire_button = read_button1();
    if (fire_button && !game->prev_fire_button) {
        LATC |= 0x80;
        reset_game_state(game);
        game->current_state = STATE_PLAYING;
        game_timer_reset();
        game_timer_start();
        reset_led_timer_tracker();
        game->prev_fire_button = fire_button;
        return;
    }
    game->prev_fire_button = fire_button;
}
```

Figure 13.) STATE_IDLE Code

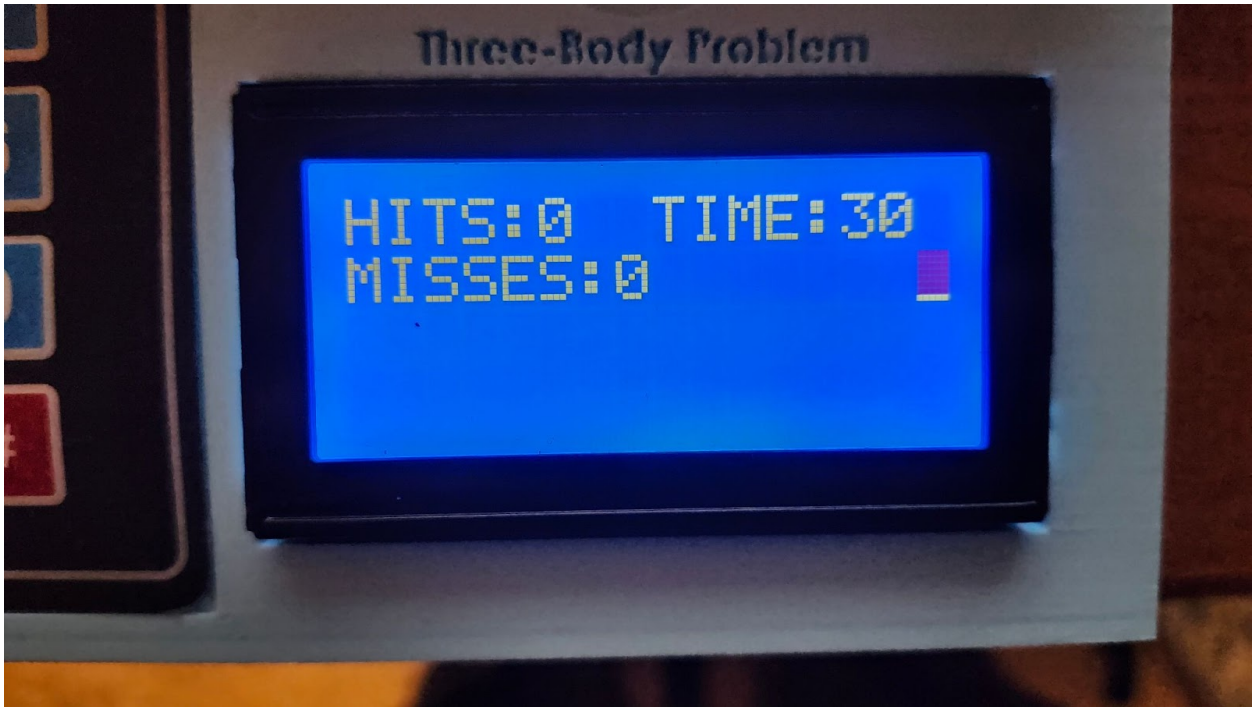

Figure 14.) STATE_IDLE LCD Screen

STATE_PLAYING Handler:

This is the main game loop. The handler continuously reads the elapsed seconds from Timer 0 via `game_timer_get_seconds`, calculates remaining time, and updates the LCD display. It then calls three critical functions in sequence: `update_led_targets` to handle LED sequencing, `handle_servo_movement` to process keypad input, and `handle_fire_button` to detect hits/misses. When the timer reaches 30 seconds or the timer stops running, the state transitions to `STATE_GAME_OVER`.

```
void state_playing_handler(GameState *game) {
    uint8_t elapsed = game_timer_get_seconds();
    uint8_t remaining = (elapsed >= 30) ? 0 : (30 - elapsed);

    game->time_remaining = remaining;
    update_lcd_game_screen(game);

    if (elapsed >= 30 || !game_timer_is_running()) {
        game->current_state = STATE_GAME_OVER;
        game_timer_stop();
        all_leds_off();
        return;
    }

    update_led_targets(game);
    handle_servo_movement(game);
    handle_fire_button(game);
}
```

Figure 15.) STATE_PLAYING Code

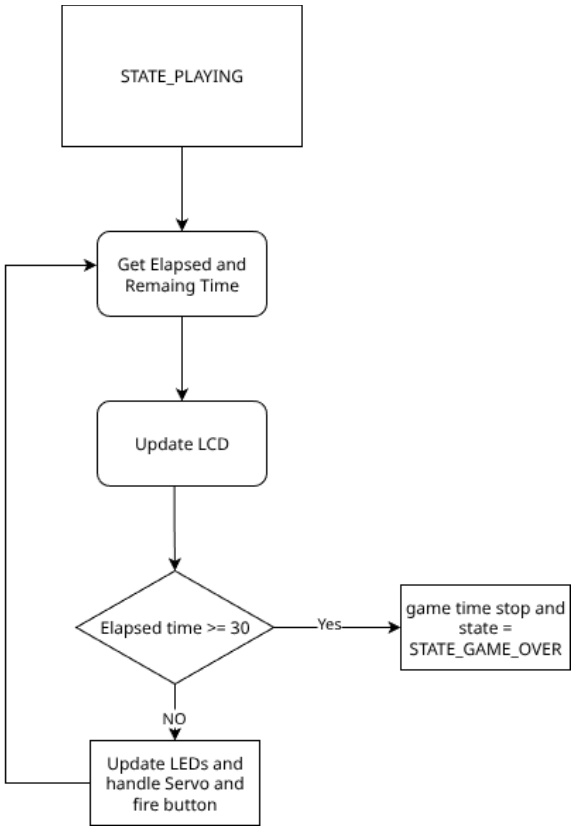

Figure 16.) Flowchart for STATE_PLAYING

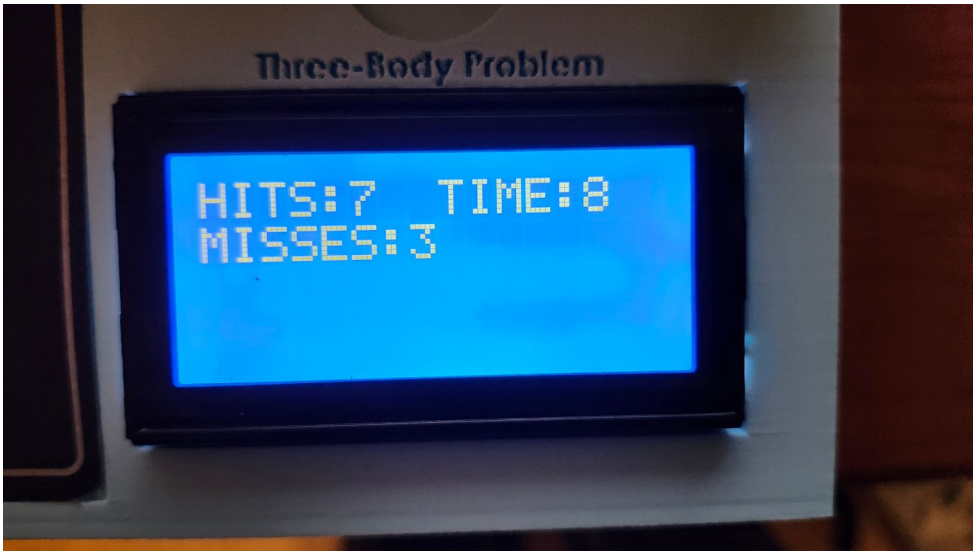

Figure 17.) STATE_PLAYING LCD Picture

STATE_GAME_OVER Handler:

The game over state displays the final score and allows the player to press the fire button to play again. If fire is pressed, it transitions back to `STATE_IDLE` (not `STATE_INIT`, so the splash screen is skipped). The `time_remaining` is forcibly set to 0 to ensure clean LCD output on the next state.

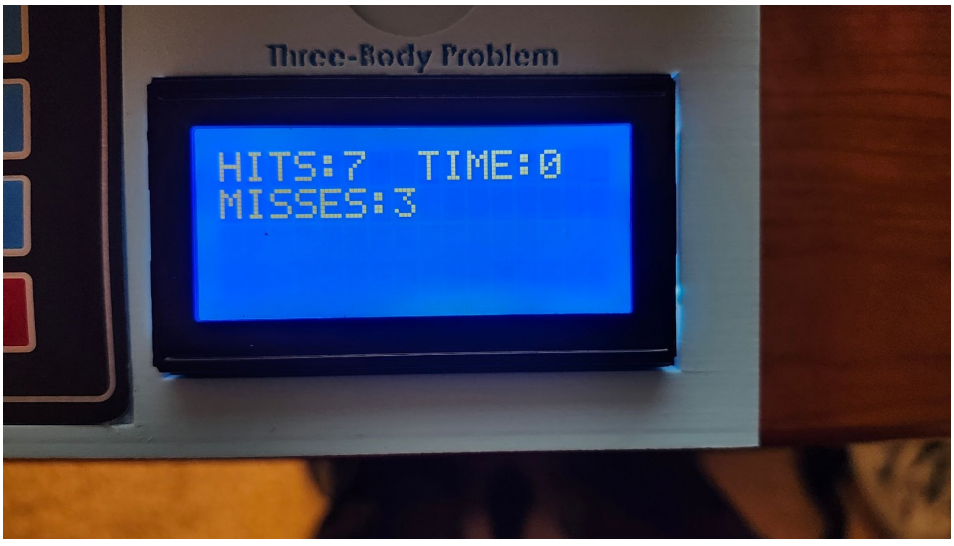

Figure 18.) Game Over Screen LCD

Input Handling Details

The button input uses rising edge detection by comparing the current button state with the previous state stored in `game->prev_fire_button`. The button hardware uses inverted logic where pressing returns 0 and releasing returns 1. The keypad scanning uses a 10-microsecond debounce delay and open-drain columns to detect key presses. The keypad value function maps column and row bit patterns to key numbers 1-12

Reset and Initialization

The `reset_game_state` function is called when transitioning from IDLE to PLAYING, and also when restarting from GAME_OVER. It zeros hits and misses, resets the LED sequence step and timer to 0, sets current_led to 3 (no LED), turns off all LEDs, and selects a random new LED mode.

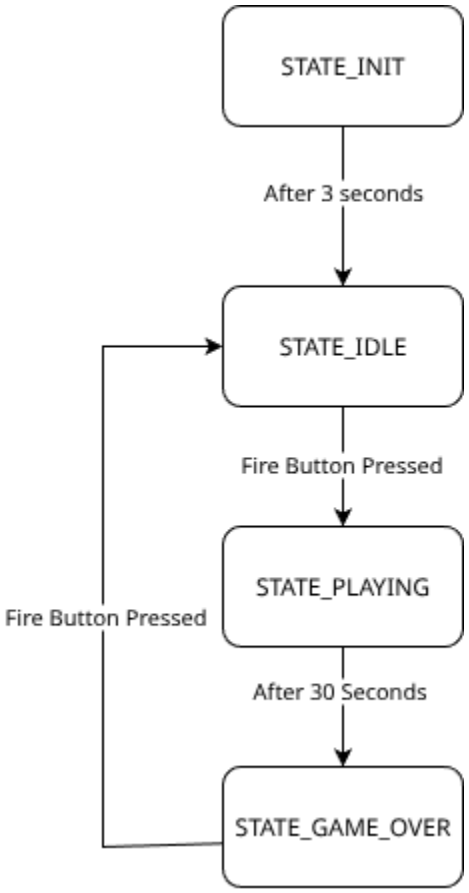

Figure 19: Game State Transition Diagram

Extra Credit

The next sections will be the extra credit portions. These will hopefully be rather short. The extra credit sections will be the GUI menu, tests section and Simon says. I'll address these sections in their respective previously designed order, let's get started on the GUI.

GUI:

This is rather simple, it consists of two parts, the main menu and "#" main menu transition. The main menu is the first screen to be executed, it preseeds the target game states. The main menu consists of 3 options and a disclaimer. The three options are "game", "simon" and "test". The `game` moves you to the target game initialization. `Simon` moves you into the simon says initialization, this will be discussed later. Lastly, the `test` moves you to a testing environment, this will also be discussed later. A disclaimer at the bottom tells the user that pressing the "#" button on the keypad brings you back to the main menu. This allows a user to go from the target game back to the main menu if needed. A figure showing the main menu screen is below followed by a figure of a GUI workflow diagram.

Figure 20.) Main Menu LCD Screen

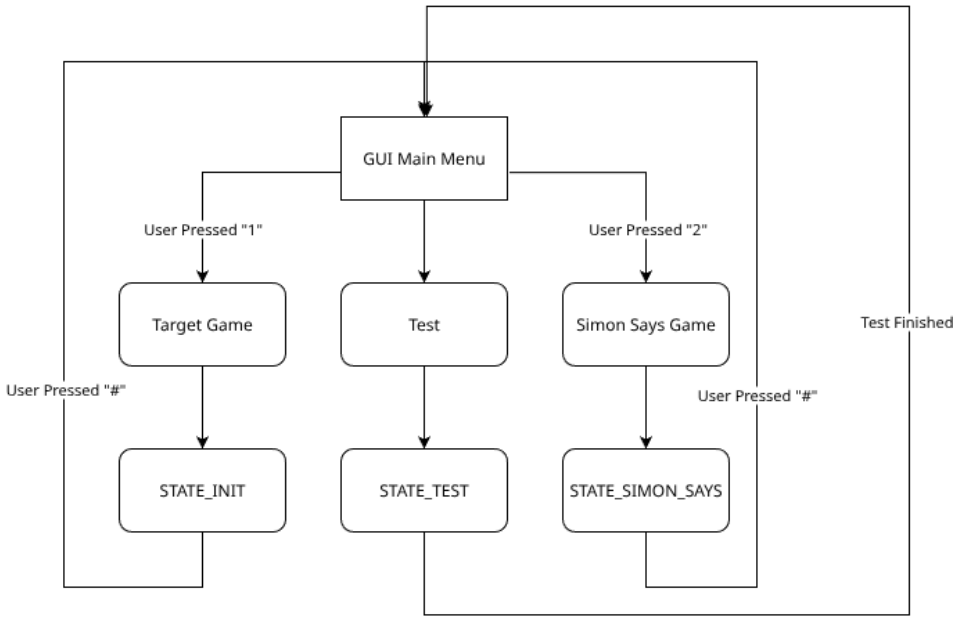

Figure 21.) GUI Workflow Diagram

To let the user go back to the main menu, the "#" code is simply added as an option to the states where a keypad input is being read. If the "#' button is pressed, it forces the game state to the `STATE_MENU`, an example of this in the code is below.

```
keypad key = numpad_press();
    if (key.pressed) { // if pressed, check for #
        uint8_t key_num = numpad_value(key.cols, key.rows);
        if (key_num == 12) {  // if # is pressed go back to main menu
            servo_centered = 0;
            all_leds_off();
            game->current_state = STATE_MENU;
            return;
        }
    }
```

Figure 22.) Code To Bring User To Main Menu When "#" Is Pressed

Tests:

The test section is an abomination of all the commented out code I used to test the individual components. To ensure that my components will work on the demonstration, I decided to uncomment these tests and move them into their own separate file. The test.c/.h contains this code.

It starts with a servo+keypad test, then moves to a LED test and lastly a speaker test. The LCD is displaying the different states and the button is never tested. After the speaker test, the test

section returns the user back to the main menu. A state diagram of the test code is shown below.

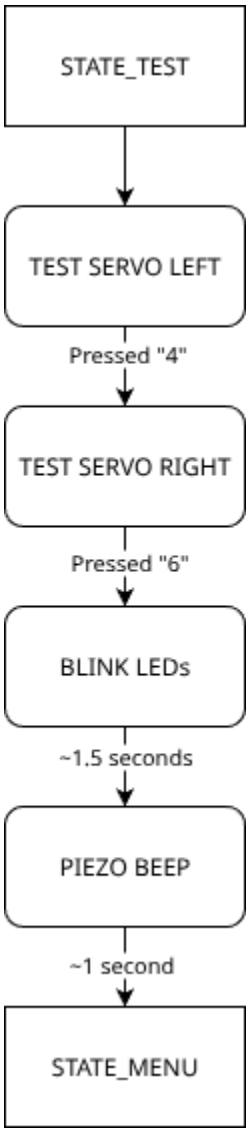

Figure 23.) Test Section Workflow Diagram

Simon Says

This section will be a little long. All the Simon says code is contained in the simon.c/.h file. The rules Simon says are below in a numbered list.

1. Every round a random sequence of LEDs is selected
2. At random “fake LED” will occur 50% of the time
    a. A fake “LED” is denoted with a noise from the speaker
3. The user will repeat the sequence but not include any fake LEDs
    a. They use the keypad to move the servo and the button to select a LED

4. If the user gets the sequence correct, the LEDs will blink twice with a speaker sound as well
    a. When a user gets a sequence right, the length of the next sequence will increase by 1 until the length is 20. (highest I ever got is 5)
5. If the user gets the sequence wrong, the red LED will blink twice with a low speaker sound
    a. LCD will display a game over screen with final score

Similar to the target game, I organized Simon's states into a predefined structure. This structure would move the game through all its states. This game state structure is shown below.

```
typedef enum {
    SIMON_SHOWING_SEQUENCE, // Displaying the sequence to memorize
    SIMON_WAITING_INPUT,  // Waiting for player input
    SIMON_SUCCESS, // Player completed sequence correctly
    SIMON_FAILURE, // Player got it wrong
    SIMON_GAME_OVER // Game ended
} SimonState;
```

Figure 24.) Simon Says States

Next, another structure to hold a game instance of simon says was created. This structure holds the variables that are required for the game to work properly. This structure would then be manipulated by functions inside of simon.c. Since some of the variables in simon says are the same, particularly the buttons, they employed static to ensure their scope doesn't leave simon.c. The Simon says structure is below.

```
typedef struct {
    uint8_t sequence[SIMON_MAX_SEQUENCE]; // The LED sequence (0=Green, 1=Red,
2=Yellow)
    uint8_t fake_index; // Which index in sequence is the fake (beep) LED
    uint8_t sequence_length; // Current sequence length
    uint8_t current_step; // Current step in showing/input
    uint8_t player_step; // Player's current input step
    uint8_t score; // Player's score (rounds completed)
    SimonState simon_state; // Current simon game state
    uint8_t sequence_shown; // Flag: has sequence been shown this round?
} SimonGame;
```

Figure 25.) Simon Game Structure

The Simon says game isn't too hard to make, it takes a lot of the logic from the target game. The most significant change is the LED sequence logic. In Simon says, it will generate an array of

LED positions at random then randomly create a fake LED. This logic isn't very verbose, the code for it is shown below.

```
void simon_generate_sequence(SimonGame *simon) { // randomly generate simon
sequence
    // Generate random sequence of LEDs (0=Green, 1=Red, 2=Yellow)
    for (uint8_t i = 0; i < simon->sequence_length; i++) {
        simon->sequence[i] = rand() % 3;
    }

    // Add a fake LED with beep (50% chance if sequence length > 2)
    if (simon->sequence_length > 2 && (rand() % 2) == 0) {
        // Pick a random position for the fake (not first or last)
        simon->fake_index = 1 + (rand() % (simon->sequence_length - 1));
        // Make sure fake LED is different from previous
        uint8_t prev_led = simon->sequence[simon->fake_index - 1];
        simon->sequence[simon->fake_index] = (prev_led + 1 + (rand() % 2)) % 3;
    } else {
        simon->fake_index = 255;  // No fake this round (lucky you)
    }

    // Reset for showing
    simon->current_step = 0;
    simon->player_step = 0;
    simon->sequence_shown = 0;  // Reset flag so sequence will be shown
}
```

Figure 26.) Simon Says Random Sequence Generation

Finally, with the Simon says game finished, a workflow diagram and figures of all the possible LCD states will be shown below.

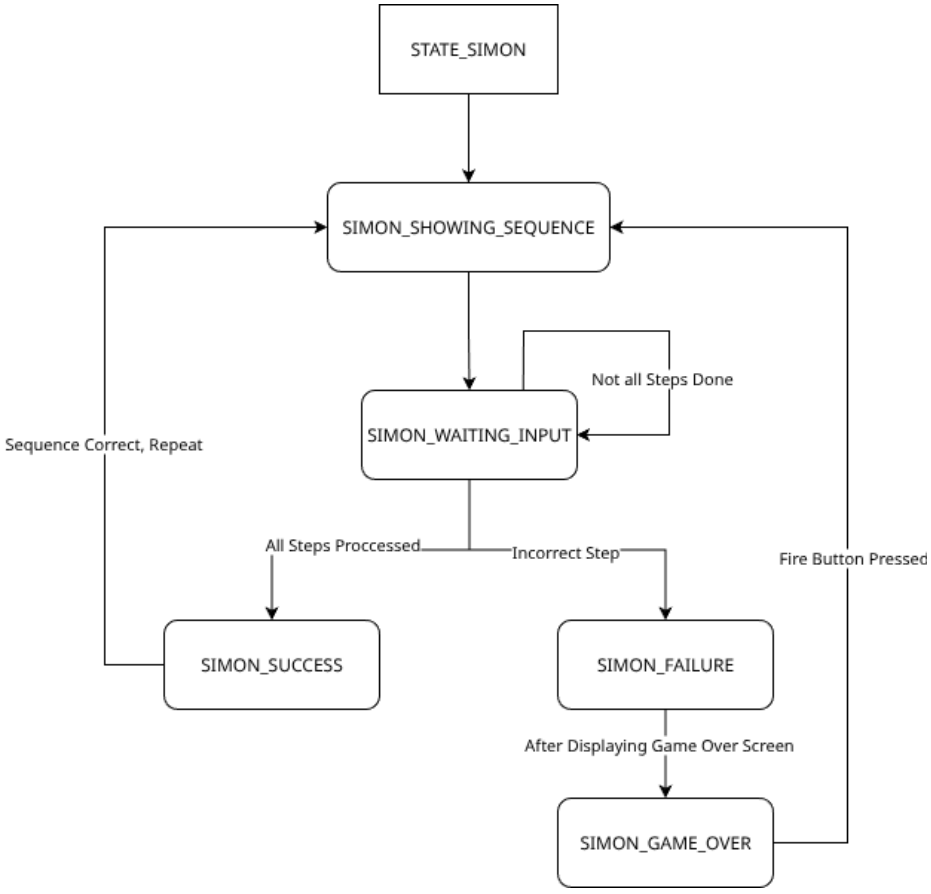

Figure 27.) Simon Says Workflow Diagram

Figure 28.) Simon Showing Sequence LCD Screen

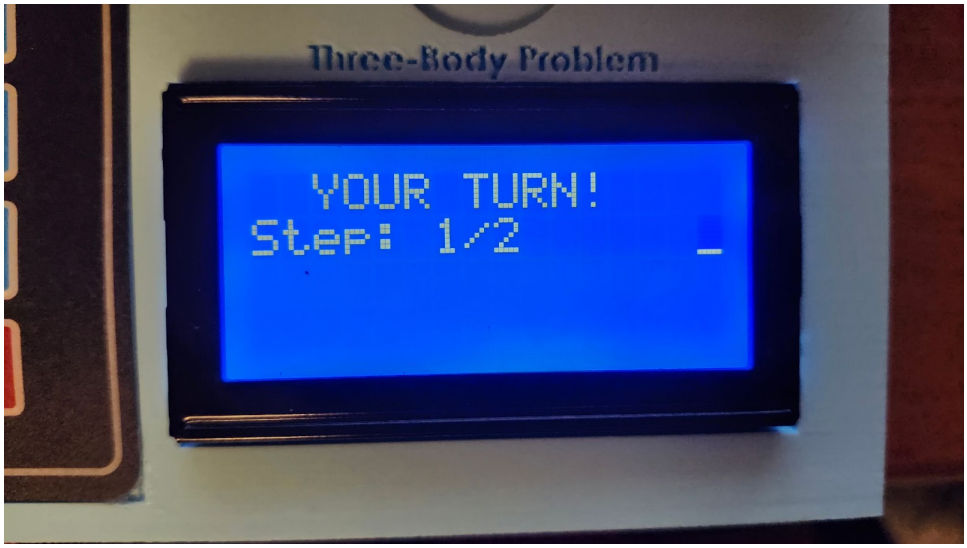

Figure 29.) Simon Waiting Input LCD Screen

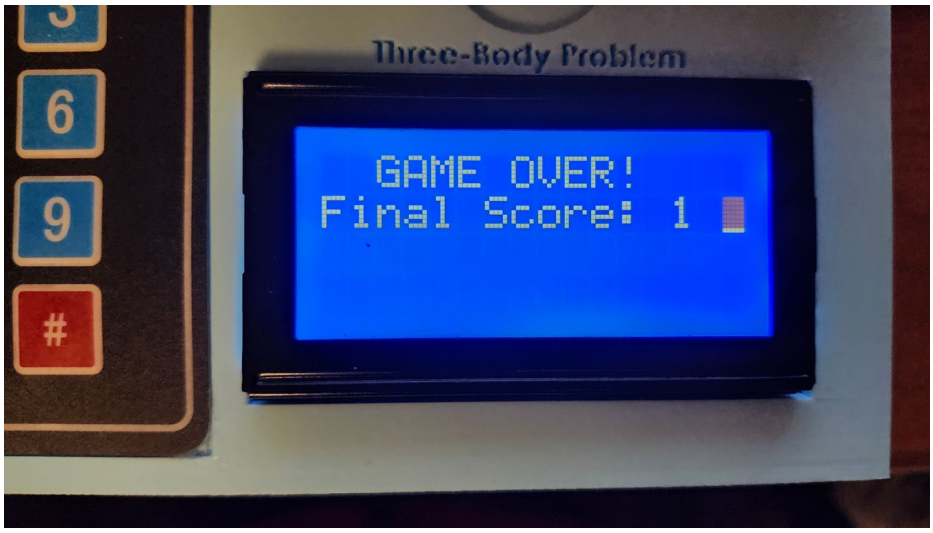

Figure 30.) Simon failure and Game Over LCD Screen

Results

This section will mainly go over a holistic workflow of this project. It will include the holistic file tree and state diagrams. I'll start with the file tree, with brief explanations of what's contained inside of each finalized file then end it with a state diagram

File Tree + Organization:

```
code/
├─── button.c
├─── button.h
├─── gui.c
├─── gui.h
├─── hardware.h
├─── keypad.c
├─── keypad.h
├─── lcd.c
├─── lcd.h
├─── led.c
├─── led.h
├─── main.c
├─── servo.c
├─── servo.h
├─── simon.c
├─── simon.h
├─── speaker.c
├─── speaker.h
├─── state.c
├─── state.h
├─── test.c
├─── test.h
├─── timer.c
└─── timer.h
```

Figure 31.) Final File Tree

button.c/.h:

Contains the initialization and logic to process button inputs

gui.c/.h:

Contains the states the gui can output and the handling logic once the state is being processed.

hardware.h

Contains the hardware pins for the circuit components

keypad.c.h:

Contains the code to initialize and process keypad inputs for the program to read. The keypad inputs are processed and stored inside of a structure

lcd.c/.h:

Contains the code to initialize, push data, write data, manage/mutate the cursor and write strings.

led.c/.h:

Contains initialization and setting code for the leds. Furthemore, it contains the update led functions and led enum mode for the target game.

main.c

Contains the main state machine conductor and initialization for the program

servo.c/.h

Contains the initialization for the servo and functions to manage it such as duty cycle changes and getters. Furthermore, contains the array that stores all possible servo positions, from `POS_1` to `POS_5`.

simon.c/.h:

Contains the initialization for the simon says game. The initialization will reset the simon says structure which is stored in the .h file. Then the .c file contains state handlers and other key functions that allow the simon says game operate. Then one holistic handler that orchestrates the entire game.

speaker.c/.h:

Contains the initialization for the speaker and other functions to operate the speaker. These functions are employed by the state.c and simon.c to produce distinguished sound effects

state.c/.h:

Contains the target game initialization. The initialization will reset the target game structure stored in the header file. Furthermore, the state header file contains part of the program-state structure for the gui menu. The .c file contains the state handlers for each state the target game can inherit. Other handlers are created to handle inputs then the states handle how these inputs mutate the game structure.

test.c/.h:

Contains code that tests all hardware functions for the project.

timer.c/.h:

Contains timer function and initialization. Notable timers and features here are Timer 0 for target game, Timer 1 for delays and the interrupt for the project. The interrupt handles Timer0 to ensure only 30 seconds of game play passes

State Diagram:

Next a final state diagram depicting all possible states the machine will be shown. Underneath the state names are the outputs of each state.

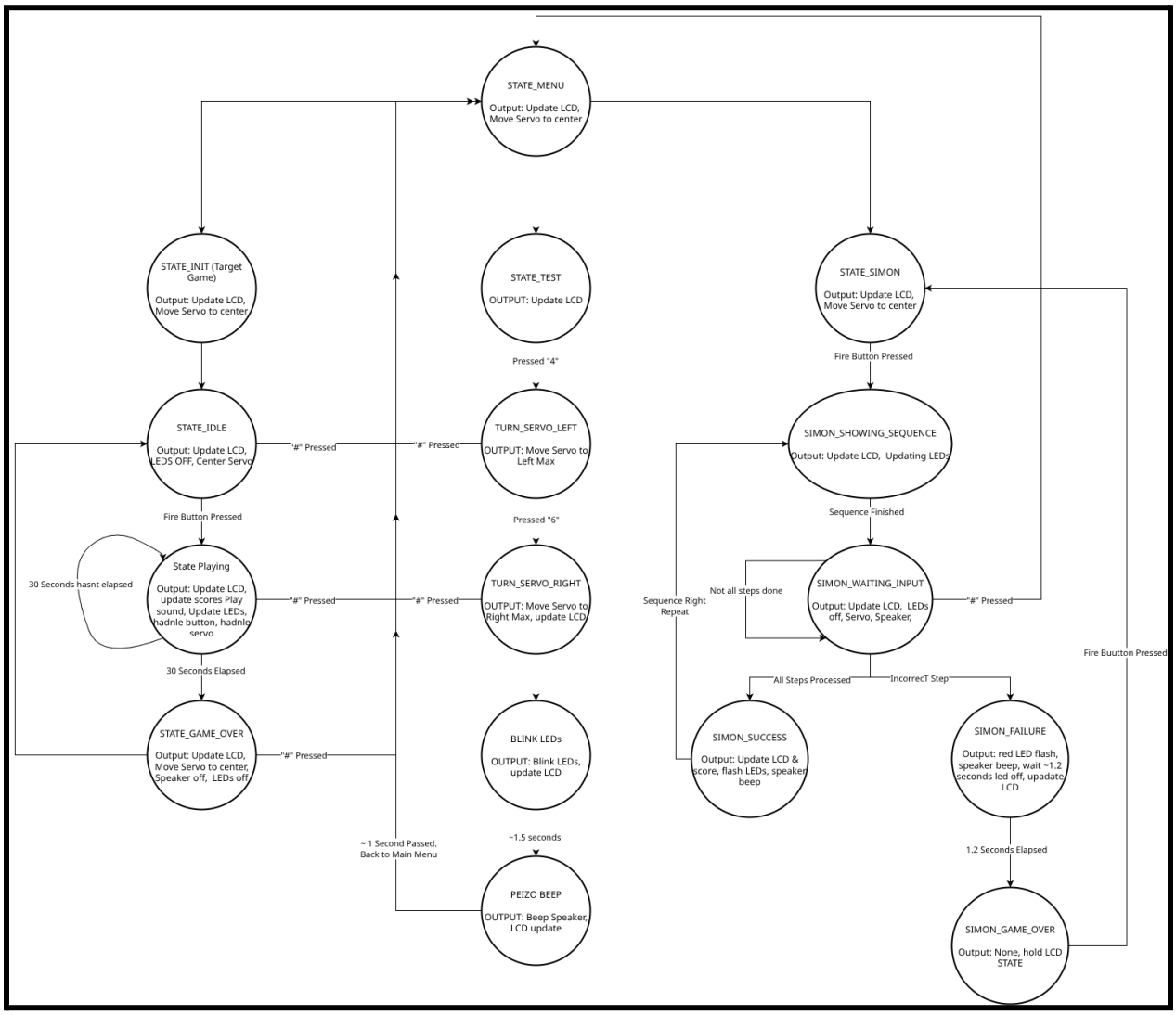

Figure 32.) Final State Diagram for Entire Program

Conclusion

Overall this project went really well. I am incredibly proud of my code. For some reason, I spent a long time ensuring my code was organized and structured. I really wanted to employ structures and enumerators to apply a layer of abstraction to the actual program. Furthemore, I like how I split everything into their own respective file. By doing this approach, I was able to single out any issue and solve it easily.

During this lab, I ran into a variety of issues. A lot of these issues were already discussed in the chronological procedure. The largest bug I ran into was with Timer0, I should have read through the LCD code more thoroughly to spot that Timer0 was being used as a delay. Since the LCD and game were both written into Timer0, it caused a sort of race condition which resulted in the game timer not counting down. To solve this I used the debug LED to toggle every interrupt to, then I noticed that it was permanently on indicating it wasn't being started. This then led to me noticing the timer0 delay and changing the code which resulted in the timer working.

I ran into other issues, however if I were to discuss them this report would already be way longer than it already is. The number one debug method I used was the onboard LED. I used it to ensure states were benign transitions, inputs were being read and a lot more. By employing the onboard LED I solved nearly all the problems. One last problem I had was solved with Professor Cakamk by inspecting my circuit, he noticed my mosfets were in the incorrect position.

Overall I think this project went incredibly well and I am really happy with what I made, however I could have improved in a lot of places. Although the separation is very organized, in some situations it becomes very verbose. Especially in the files that manage states, they depend on a lot of pointers and obscure references to the game-structure. Thankfully, I used structs in the past which helped a lot for this lab.

Another area of improvement would be this report. I didn't have much time to make the report. I apologize if it seems rushed, to be brutally honest it is rushed. Furthermore, the reason why it's so long is because I didn't have time to double check through everything so I just added all the information to ensure I didn't miss anything. I had a very busy week with two projects due and two more during exam week. I also have 3 exams coming up as well, I took an exam during the weekend as well. I really regret that this report is going to be immature. I really enjoyed this project and I'm proud of what I made, I wish I could flesh it out even more.

I liked this class as well, I always took for granted how nice it was for hardware to just work. A lot of issues I had during labs were due to hardware and I learned a lot on how to debug them. Overall, I would say I have mixed feelings on microcontrollers. I like programming them but the

hardware and datasheets are really tough. I hope I get an A in the class. Thank you for a fun semester 🙂

Code

Located on the blackboard. The zipfile contains all the header files besides main, main will be sent as its own file. If you are confused on the structure, look at the readme.