Reaction-Time Game
Scott Pearson and Braden Miller
5/9/2025
ECE:3360 Embedded Systems

I. Introduction

a. Motivation and Background

Throughout the entire semester, we explored how critical precise timing is in the context of embedded system applications. Nearly every lab emphasized achieving accurate time control for reliable functionality. Building on this foundation, our final project centered on creating a reaction-based time game using the ATmega328P microcontroller alongside various integrated components. Our completed game provided a hands-on opportunity to apply C-based embedded programming while demonstrating key skills such as handling external interrupts, managing I2C communication with an OLED display, generating PWM signals for RGB LED control, and using EEPROM to store related data. The system highlights the importance of real-time responsiveness in embedded design, especially in interactive applications that are dependent on processing user input in a timely manner.

b. Goals/Specifications

The goal of this lab was to design and implement a fully functional, interactive reaction game featuring an OLED display, RGB LED, buzzer, and pushbutton input. The system uses interrupt-based event handling, a finite state machine for game logic, and EEPROM to store high scores across power cycles. The user is guided through various game states including a welcome screen, difficulty selection, countdown phase, reaction measurement, and final score display with game difficulty influencing response timing thresholds.

II. Implementation

a. Overview

As mentioned previously, our project was designed around our ATmega328P microcontroller. It was centered on creating a state-driven reaction-time game. Our system integrated several key hardware components. These specific components involved an SSD1306 OLED display to display the game's visual output, an RGB LED for real-time color feedback using PWM, a pushbutton input for user interaction via external interrupts, a buzzer for sounding wrong clicks, and EEPROM memory for storing high scores across difficultly levels. Our system's software is modularly structured around a finite state machine (FSM). Our FSM orchestrated the flow of our game system through a collection of changing stages. These stages included the startup, difficultly selection, countdown, reaction measurement, result display, game over, and losing. Communication with the OLED was handled through I2C by utilizing an external driver library, while color transitions on the RGB LED were driven via Timer1 and Timer2 PWM channels. Our design process emphasized incremental integration since each hardware module was independently tested prior to being implemented into the larger state machine. Our design structure ensured that our system remained stable and maintainable. With

this, we were able to have an easier time enabling dynamic interaction between asynchronous user input and timed system responses.

b. Hardware description

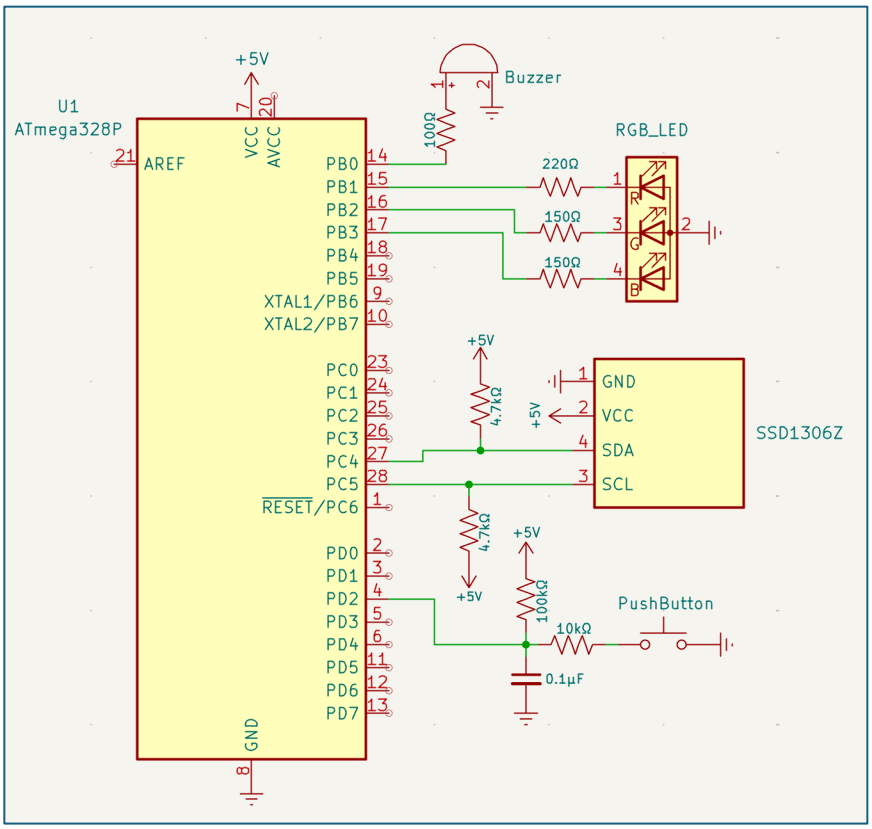

Figure 1.0: Circuit as Implemented.

Before building any hardware, we drew a schematic to provide a solid blueprint for us to follow (see Figure 1.0). The core of the system is built around the ATmega328P microcontroller, which coordinates all input, output, and timing functions for our reaction game. A SSD1306 OLED display communicates with the microcontroller over I2C protocol via PC4 (SDA) and PC5 (SCL). Both connections were each pulled up with 4.7kΩ resistors to ensure reliable signaling. Visual feedback is provided through a common cathode RGB LED driven by PB1 (red), PB2 (green), and PB3 (blue), each with current-limiting resistors (see Figure 1.1) (220Ω for red, 150Ω for green and blue). A small buzzer is connected to PB0 through a 100Ω resistor for audio output purposes during early presses or state transitions.

$$R_{\text{red}} = \frac{5\,\text{V} - 2\,\text{V}}{0.015\,\text{A}} = 200\,\Omega \rightarrow \boxed{220\,\Omega}$$

$$R_{\text{green}} = \frac{5\,\text{V} - 3\,\text{V}}{0.015\,\text{A}} = 133\,\Omega \rightarrow \boxed{150\,\Omega}$$

$$R_{\text{blue}} = \frac{5\,\text{V} - 3\,\text{V}}{0.015\,\text{A}} = 133\,\Omega \rightarrow \boxed{150\,\Omega}$$

Figure 1.1: RGB LED Resistor Calculations.

User input is captured via pushbutton connected to PD2 (INT0), configured with an external pull-up resistor (10kΩ) and a debouncing capacitor (0.1µF) to ground. This configuration ensures a clean, noise-free signal and allows for interrupt-based detection of button presses. Power for all system components is supplied at +5V, with common ground shared across the system. Reference datasheets for the ATmega328P, SSD1306 display, and RGB LED were regularly consulted to ensure proper pin configurations and electrical characteristics.

c. Software description

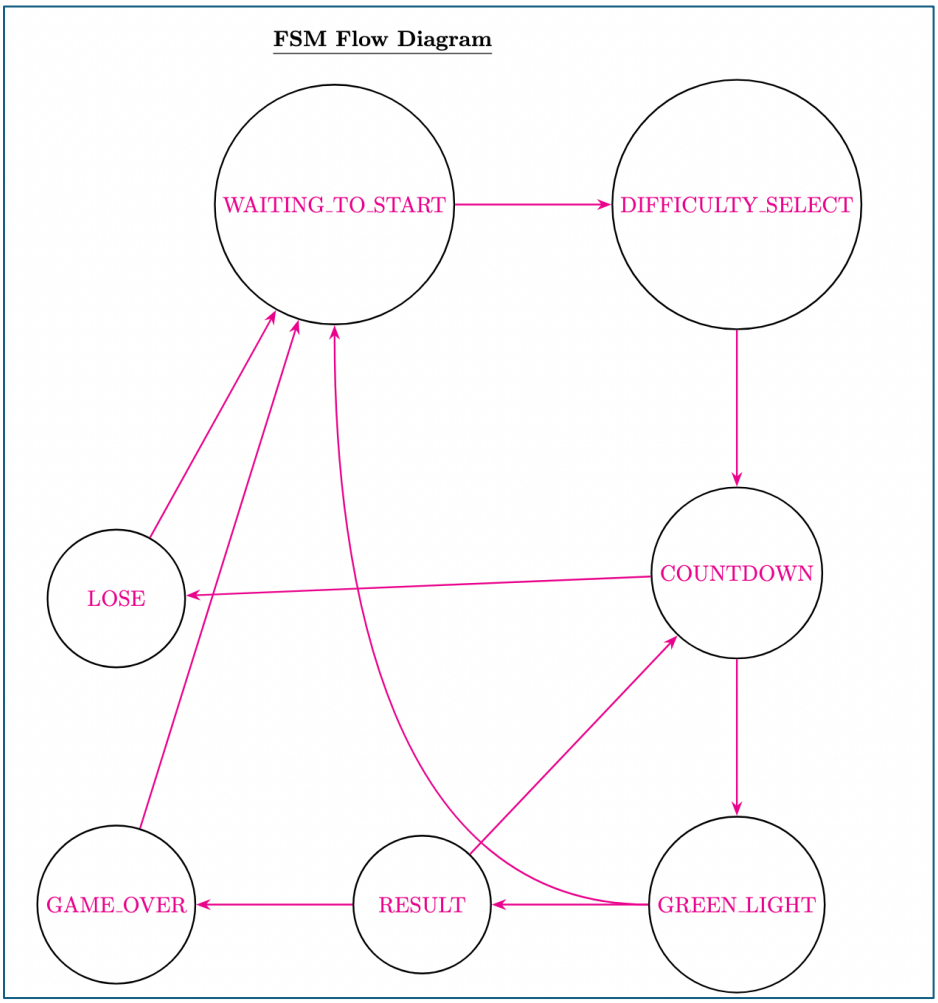

Figure 1.2: FSM Flow Diagram (created in LATEX)

The software is organized around a finite state machine (FSM) that governs the flow of the game (see Figure 1.2). The main loop continuously checks the current state and executes the related logic for that given state. Transitions between states are triggered by user input, elapsed time, or internal flags set via interrupts. The main “game flow” of the program can be seen in the lower right of the diagram - the loop between countdown, green light, and result. The sequence of these states represents a game being played: the player is waiting in the countdown stage, the player pushes the button when the light turns green, they get the results of their selection, and they are taken back to a countdown for the next level.

In reference to each of the states in the FSM, their associated responsibilities are as follows:

1. WAITING_TO_START: Initializes the game and waits for the user to press the button to begin.
2. DIFFICULTY_SELECT: User can cycle through and choose a difficulty level.
3. COUNTDOWN: Based on the difficulty selection, a randomized delay is introduced to test the user's reaction time. If pressed too early, the game transitions to LOSE. Otherwise, it processes the input as valid and transitions users to GREEN_LIGHT.
4. GREEN_LIGHT: The user must press the pushbutton as quickly as possible when the green signal appears. Reaction time is measured once the button is pressed. Depending on the round number, the game either transitions to RESULT (to display timing) or WAITING_TO_START if an invalid press occurs.
5. RESULT: The measured reaction time from the user is displayed and optionally stored in EEPROM if it's a new high score. The game advances to either the next COUNTDOWN if there are more rounds to be played or to GAME_OVER.
6. GAME_OVER: After successful completing five rounds, a summary screen is displayed. The system then returns to WAITING_TO_START.
7. LOSE: This stage is triggered by a premature button press during the countdown. It displays a message and resets the game.

The major functions of the game are as follows:

1. display_message(): This method handles all screen output using the OLED via I2C.
2. set_rgb_color(): This adjusts the RGB LED using PWM based on the game state (e.g., red = early press, green = go).
3. measure_reaction_time(): This method tracks the user's speed with millis() or internal timing loops.
4. update_high_scores(): This is responsible for reading and writing EEPROM data for persistent storage of best times.

For Interrupt Service Routines (ISRs), External Interrupt INT0 (PD2) was used to detect button presses asynchronously. It was responsible for setting flags, like button_pressed and button_released, for the main loop to process without blocking. Debouncing for our system was handled in software with short delays or state checks to prevent false triggers. In the context of EEPROM access, no interrupt-based EEPROM was used. Instead, it was accessed synchronously during transitions between rounds. The FSM-based software architecture we used ensured clarity, modularity, and responsiveness for handling asynchronous user inputs and present-state timing logic.

III. Experimental Methods

To begin our testing for experimental purposes, we connected the SSD1306 OLED display to the ATmega328P via the I2C interface. After we ensured our wiring was set up properly, we moved on to downloading compatible display and I2C libraries. Our search for valid libraries took longer than expected mainly because of configuration issues and extremely sophisticated libraries that were hard to comprehend. However, we were eventually able to get our hands a very simple, yet effective library to help display a simple "Hello, World" message to verify that

communication was set up. This display message function became the backbone of our project in regards to showing game information to the user.

With our display operational, we shifted focus on designing the software for our system. We created the FSM diagram (see Figure 1.2) to serve as the foundation of our game. This visualization helped guide the flow of our game logic and allowed us to implement new changes without disrupting previous completed work.

As we implemented the game bit by bit, we added and tested each state one at a time. Whenever new functionalities were added, we would test our system to ensure previously working features remained stable. For instance, after integrating the countdown logic, we tested both normal and early button presses to confirm correct state transitions to either the green light or lose state.

For testing EEPROM functionality, we repeatedly ran full game cycles at each difficulty level to confirm that high scores were correctly saved and loaded across power cycles. We verified that the scores were saved and written to the correct memory locations by checking EEPROM content after resets.

IV. Results

The final implementation of our reaction-game functioned exactly as we had intended. We were able to successfully integrate our hardware and software components into a cohesive interactive system. The OLED display properly rendered in all the interface elements, including welcome screens, difficulty selection, countdown prompts, reaction time feedback, and the final summary page for high scores stored in EEPROM. Each game state had smooth transitions and accurately corresponded with user input and/or internal timing logic. The RGB LED provided aesthetic visual cues for the different state indications of the game —red for countdown, green for go, blue for level results, and orange for game-over phases—using PWM control.

User input was captured via button presses. These presses were consistently registered through the external interrupt configuration. The buzzer was activated upon invalid inputs during the countdown phase, confirming real-time interrupt handling. Reaction times were correctly calculated and displayed with millisecond accuracy. New top scores were successfully stored in EEPROM and correctly retrieved on subsequent runs, demonstrating persistent memory functionality.

To support these claims, we have included screenshots of the OLED screen at various stages of the game, photos of the hardware in operation, and images showing EEPROM-based high score tracking across power cycles. Together, these visuals validate the project's complete functionality and reliability under real-world conditions

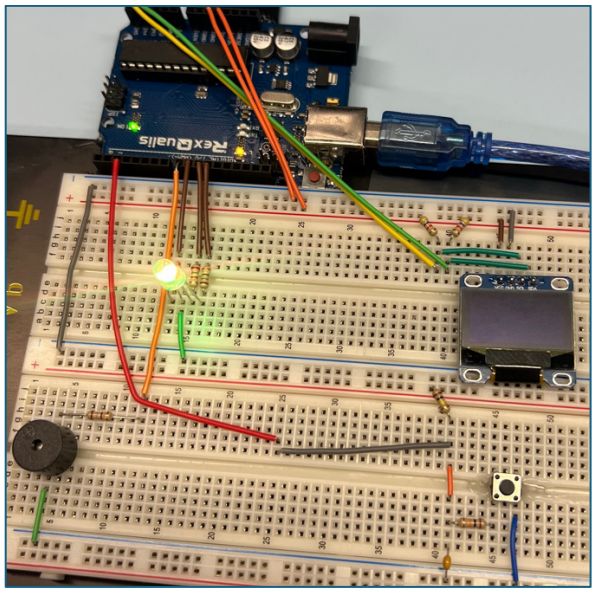

Figure 1.3: Completed Hardware

This figure shows our base hardware as implemented in our schematic. Drawing the schematic out first made wiring our components not too challenging.

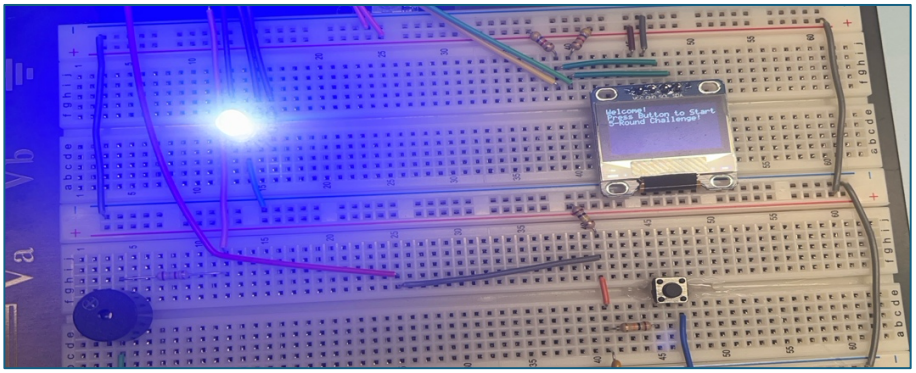

Figure 1.4: Welcome Screen

This is the welcome screen that the user sees upon powering up the system. After the screen fully loads, the user presses the pushbutton to go on to select a difficulty level for a new game.

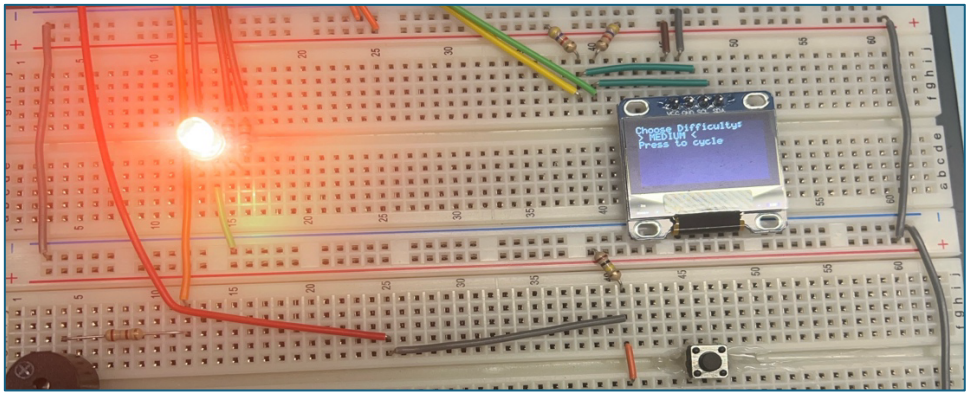

Figure 1.5: Difficulty Selection Screen

This is the screen users see when selecting a difficulty. They can cycle through difficulties by tapping the pushbutton. When they find out what difficulty they want, they hold the pushbutton until the loading is complete (a message will appear under the “Press to cycle” that will show the loading percentage to transition to the start of the game. This percentage is updated in real-time).

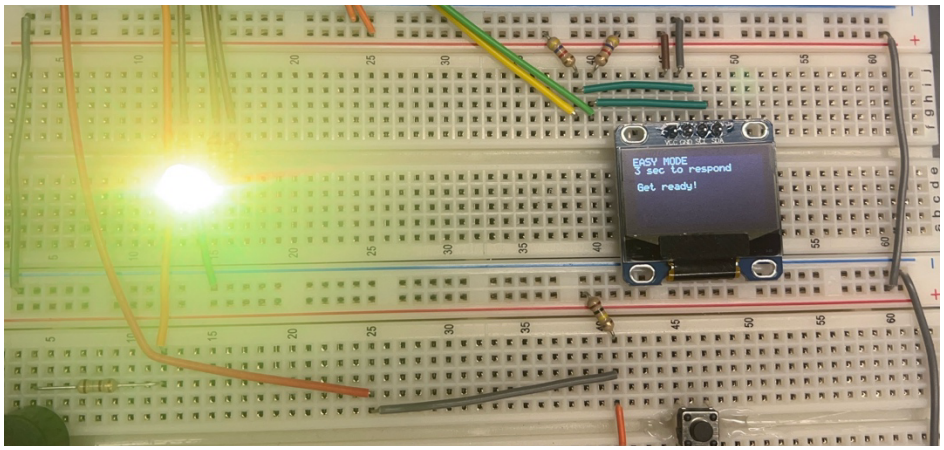

Figure 1.6: Easy Mode Selected – Game About to Start

In this frame, the user has selected and loaded a game for “EASY” mode. The game will then start, and the user will have to press the pushbutton when the RGB LED transitions from red to green.

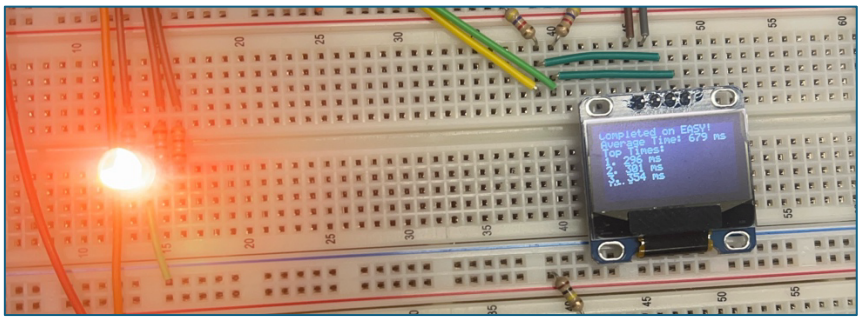

Figure 1.7: Game Over Screen – Game Completed on Easy Mode

In this frame, the user has successful completed the game on “EASY” mode. As can be seen, the top 3 scores are displayed. These scores were stored in EEPROM. Since the user’s score wasn’t higher than the previous top 3, their score will not replace the top scores.

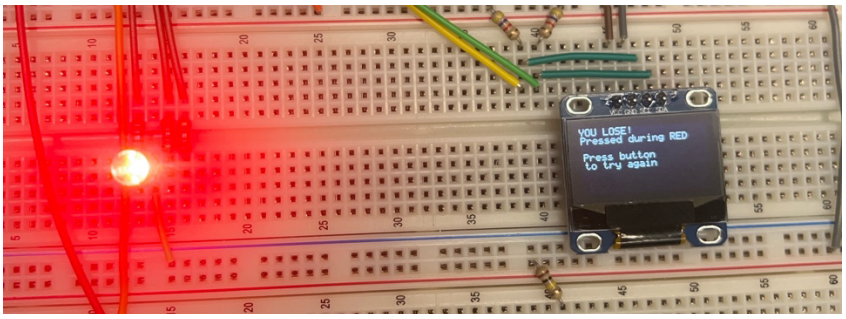

Figure 1.8: Lose Screen – Buzzer Sounded due to Invalid Input

In this frame, the user accidently pressed the pushbutton during an invalid timeframe – either during the countdown or they missed the greenlight time frame. They are prompted to try again by selecting the button.

V. Discussion of Results

As previously stated, the device performed exactly as we had intended. The system performed reliably and responsively throughout testing and demonstration phases of the project. Our reaction timing was accurate, transitions between states were seamless, and both visual and auditory feedback components properly functioned as expected. Our system was able to effectively handle asynchronous inputs. With this, real-time responsiveness validated our use of interrupts and PWM for the embedded interactive game we constructed.

All our original design goals were met. Our game allowed users to select a chosen difficulty level, respond to randomized timing windows, view their scores for reaction times, and retain high scores across power cycles via EEPROM. The OLED display, RGB LED, pushbutton, and buzzer all integrated perfectly with the ATmega328P microcontroller. Each hardware-software interaction consistently performed during our testing phases.

In terms of alternative implementation strategies, we could have used a much larger SSD1306 OLED display so we could display more information. Also, we thought about possibly using an RFID scanner to hold usernames so players could maintain usernames and personal scores.

Current limitations of our system in its present stare stem from our use of a fixed random seed for countdown timing. This reduces variability across power cycles and the reliance on software debouncing. Additionally, EEPROM writes are currently performed after each round. In doing this, it has the potential to shorten the lifespan of the EEPROM if the game is played excessively.

To improve the system further, future iterations could incorporate any of the following ideas:

1. RFID scanner to hold usernames in EEPROM to display leaderboards.
2. Multiple games at once using multiple buttons and LEDs.
3. Additional game modes, animations, or sound feedback to enhance the user experience.

VI. Conclusion

Throughout this project, we were able to successfully demonstrate the integration of multiple hardware and software subsystems within an embedded environment. We did our best to incorporate aspects and lessons learned from each of the labs leading into this final project. In doing so, we developed a fully functional reaction-time game that utilized an ATmega328P microcontroller to coordinate input from a pushbutton, visual feedback via an OLED screen and RGB LED, and audio output through a buzzer. Our software architecture, based on a finite state machine, allowed for clear, detectable, and maintainable state transitions as well as responsive real-time interactions. Additionally, persistent data storage using EEPROM enabled the system to keep track of and retain top scores across resets. These aspects lead to us fulfill all initial design objectives.

The work we completed in this project sheds a light on several core concepts in embedded systems development. These include I2C communication, PWM configuration, external interrupts, software debouncing, and memory management. In broader terms, the success of this project illustrates how embedded platforms can be utilized to construct interactive, user-focused systems that are both robust and extensive. The modular design we established and stuck with, alongside the clearly defined state structure, provided us with a strong foundation for future enhancements or adaptation into other timing-based applications.

VII. Acknowledgements

Our success and accomplishments would not have happened without the help of Professor Beichel and the wonderful TAs for the course. The knowledge and lessons learned in lectures directly translated to our success in the lab. Whenever we were stuck on any hardware issues, needed help debugging software, or just had any questions in general, every TA was more than happy to take time to help solve our problems. For this, we would like to thank each one of the staff members for making this extremely difficult course and project a little less stressful.

VIII. References

https://github.com/prestonsn/AVR-OLED-SSD1306-IIC-DRIVER

https://github.com/prestonsn/AtmegaXX-I2C-Library

“Internal Timers of Arduino.” *Arduino Project Hub*, projecthub.arduino.cc/Marcazzan_M/internal-timers-of-arduino-6c0f66. Accessed 11 Mar. 2025.

Lecture slides (Lab4_ES_S25_f.pdf, ES25_07_LCD.pdf, ES25_08_Interrupts.pdf, ES25_Projects.pdf, ES25_EEPROM.pdf, ES25_SPI_I2C.pdf). Reinhard Beichal

ARDUINO_V2.pdf datasheet

https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf

IX. Appendix A: Source Code

```
/**
 * Reaction Time Game for AVR Microcontroller with OLED Display and RGB LED
 * ----------------------------------------------------------------------
 * This program implements a "Red Light, Green Light" reaction time game:
 * - Player waits for the light to turn from red to green
 * - When green light appears, player must press button as fast as possible
 * - Game has multiple difficulty levels and saves high scores in EEPROM
 * - Tracks reaction times over 5 rounds and calculates average
 *
 * Hardware Requirements:
 * - AVR Microcontroller (ATmega328P) running at 16MHz
 * - RGB LED for visual feedback
 * - Pushbutton connected to PD2 (INT0)
 * - Buzzer connected to PB0
 * - SSD1306 OLED Display connected via I2C
 *
 * Authors:
 * Braden Miller and Scott Pearson
 */

// Define CPU clock frequency for delay calculations
#define F_CPU 16000000UL  // 16 MHz clock speed

// Include necessary libraries
#include <avr/io.h>          // AVR I/O operations
#include <avr/interrupt.h>    // For interrupt handling
#include <avr/eeprom.h>       // For storing high scores in non-volatile memory
#include <util/delay.h>       // For delay functions
#include <stdlib.h>           // For random number generation
#include "i2c.h"              // I2C communication for OLED
#include "SSD1306.h"          // OLED display driver

/**
 * PIN MAPPING DEFINITIONS
 */
#define BUZZER_PIN 0   // PB0 - Output for buzzer control
#define BUTTON_PIN 2   // PD2 - Input for button (connected to INT0)

/**
 * GAME STATE DEFINITIONS
 * These define the different states the game can be in
 */
#define STATE_WAITING_TO_START 0  // Initial state, showing welcome screen
#define STATE_DIFFICULTY_SELECT 1  // Player selecting game difficulty
#define STATE_COUNTDOWN        2  // Red light phase, waiting random time
```

```
#define STATE_GREEN_LIGHT      3  // Green light phase, measuring reaction time
#define STATE_RESULT           4  // Showing results of current round
#define STATE_GAME_OVER        5  // Game completed, showing final results and high scores
#define STATE_LOSE             6  // Player pressed during red light, game over

/**
 * DIFFICULTY LEVEL DEFINITIONS
 * Each difficulty has different timeout values
 */
#define DIFFICULTY_EASY   0  // Easy mode - 3 seconds to respond
#define DIFFICULTY_MEDIUM 1  // Medium mode - 1.5 seconds to respond
#define DIFFICULTY_HARD   2  // Hard mode - 0.5 seconds to respond

/**
 * GAME CONFIGURATION CONSTANTS
 */
#define MAX_ROUNDS 5             // Number of rounds in a complete game
#define TOP_SCORES_COUNT 3       // Number of top scores to save per difficulty
#define TIMING_COMPENSATION 90   // Compensation value for timing inaccuracy (in ms)

/**
 * EEPROM MEMORY MAP DEFINITIONS
 * Defines where different data is stored in EEPROM
 */
#define EEPROM_INIT_MARKER 0xAA  // Magic number to check if EEPROM has been
initialized
#define EEPROM_INIT_ADDR   0     // Address to store initialization marker
#define EEPROM_EASY_ADDR   2     // Starting address for easy difficulty high scores (6
bytes)
#define EEPROM_MEDIUM_ADDR 14    // Starting address for medium difficulty high scores
(6 bytes)
#define EEPROM_HARD_ADDR   26    // Starting address for hard difficulty high scores (6
bytes)

/**
 * GLOBAL VARIABLES
 * These track the game state and player performance
 */
// Button state tracking
volatile uint8_t button_state = 0;        // Current button state (1 = pressed, 0 = released)
volatile uint8_t button_pressed = 0;      // Flag for button press event detection
volatile uint8_t button_released = 0;     // Flag for button release event detection

// Game state tracking
volatile uint8_t game_state = STATE_WAITING_TO_START;  // Current state of the game
uint16_t reaction_time = 0;               // Current reaction time in milliseconds
```

```
uint16_t total_reaction_time = 0;         // Sum of all reaction times for average calculation
uint16_t reaction_times[MAX_ROUNDS];      // Array to store each round's reaction time
uint8_t current_round = 0;                // Current game round (1 to MAX_ROUNDS)
uint8_t difficulty_level = DIFFICULTY_MEDIUM; // Current difficulty level
uint16_t green_light_timeout = 2500;      // Maximum time to respond in green light state (ms)
uint16_t top_scores[3][TOP_SCORES_COUNT]; // 2D array to store top scores for each
difficulty

// Flag to control buzzer in timeout situation
volatile uint8_t timeout_buzzer_active = 0;  // Flag to activate buzzer during timeout

/**
 * INTERRUPT SERVICE ROUTINE FOR BUTTON
 * Handles button press and release detection with debouncing
 */
ISR(INT0_vect) {
        // Small debounce delay to prevent false triggers from switch bounce
        _delay_ms(10);

        // Read button state (LOW when pressed due to pull-up resistor)
        uint8_t new_button_state = !(PIND & (1 << BUTTON_PIN));

        // Detect rising edge (button press)
        if (new_button_state && !button_state) {
                // Only set button_pressed flag if we're not in the results screen
                // And don't register button presses during game over hold detection
                if (game_state != STATE_RESULT) {
                        button_pressed = 1;  // Set the flag to indicate button was pressed
                }
        }

        // Detect falling edge (button release)
        if (!new_button_state && button_state) {
                button_released = 1;  // Set the flag to indicate button was released
        }

        button_state = new_button_state;  // Update the current button state

        // Sound buzzer if button is pressed during red light (penalty)
        // This gives immediate feedback when player makes a mistake
        if (button_state && game_state == STATE_COUNTDOWN) {
                PORTB |= (1 << BUZZER_PIN);  // Buzzer ON
        } else {
                PORTB &= ~(1 << BUZZER_PIN); // Buzzer OFF
        }
}
```

```
/**
 * EEPROM FUNCTIONS
 * These functions handle storing and retrieving high scores
 * Initialize EEPROM for first-time use
 * Checks if EEPROM has been initialized and sets defaults if not
 */
void init_eeprom() {
	// Check if EEPROM has been initialized already by reading marker
	uint8_t init_marker = eeprom_read_byte((uint8_t*)EEPROM_INIT_ADDR);

	if (init_marker != EEPROM_INIT_MARKER) {
		// EEPROM not initialized, initialize with default values (9999 ms)
		for (uint8_t diff = 0; diff < 3; diff++) {
			// Calculate starting address based on difficulty level
			uint16_t* addr = (uint16_t*)(diff == DIFFICULTY_EASY ?
EEPROM_EASY_ADDR :
					diff == DIFFICULTY_MEDIUM ?
EEPROM_MEDIUM_ADDR :
					EEPROM_HARD_ADDR);

			// Initialize top scores to 9999 (worst possible time)
			for (uint8_t i = 0; i < TOP_SCORES_COUNT; i++) {
				eeprom_write_word(addr + i, 9999);
			}
		}

		// Mark EEPROM as initialized by writing the magic number
		eeprom_write_byte((uint8_t*)EEPROM_INIT_ADDR,
EEPROM_INIT_MARKER);
	}
}

/**
 * Read top scores from EEPROM based on difficulty level
 * @param difficulty The difficulty level to read scores for
 */
void read_top_scores(uint8_t difficulty) {
	// Calculate starting address based on difficulty level
	uint16_t* addr = (uint16_t*)(difficulty == DIFFICULTY_EASY ?
EEPROM_EASY_ADDR :
			difficulty == DIFFICULTY_MEDIUM ? EEPROM_MEDIUM_ADDR
:
			EEPROM_HARD_ADDR);

	// Read top scores for the specified difficulty
```

```
	for (uint8_t i = 0; i < TOP_SCORES_COUNT; i++) {
		top_scores[difficulty][i] = eeprom_read_word(addr + i);
	}
}

/**
 * Update top scores with a new score if it's better than existing ones
 * @param difficulty The difficulty level to update scores for
 * @param new_score The new score to potentially add to top scores
 */
void update_top_scores(uint8_t difficulty, uint16_t new_score) {
	// First read current scores from EEPROM
	read_top_scores(difficulty);

	// Check if new score is better than any existing scores
	// Lower scores are better since they represent faster reaction times
	uint8_t insert_pos = TOP_SCORES_COUNT; // Default to not inserting

	for (uint8_t i = 0; i < TOP_SCORES_COUNT; i++) {
		if (new_score < top_scores[difficulty][i]) {
			insert_pos = i;  // Found position where new score should be inserted
			break;
		}
	}

	// If new score is good enough to be a top score
	if (insert_pos < TOP_SCORES_COUNT) {
		// Shift lower scores down to make room for new score
		for (uint8_t i = TOP_SCORES_COUNT - 1; i > insert_pos; i--) {
			top_scores[difficulty][i] = top_scores[difficulty][i - 1];
		}

		// Insert new score at the right position
		top_scores[difficulty][insert_pos] = new_score;

		// Calculate EEPROM address for storing updated scores
		uint16_t* addr = (uint16_t*)(difficulty == DIFFICULTY_EASY ?
EEPROM_EASY_ADDR :
					difficulty == DIFFICULTY_MEDIUM ?
EEPROM_MEDIUM_ADDR :
					EEPROM_HARD_ADDR);

		// Save updated scores to EEPROM
		for (uint8_t i = 0; i < TOP_SCORES_COUNT; i++) {
			eeprom_write_word(addr + i, top_scores[difficulty][i]);
		}
```

```
        }
}

/**
 * Initialize PWM for RGB LED control
 * Sets up timer registers for controlling color via PWM
 */
void pwm_init() {
        // Configure Timer 1 (16-bit) for PWM on OC1A and OC1B pins (red and green
channels)
        TCCR1A = (1 << COM1A1) | (1 << COM1B1) | (1 << WGM10);  // Fast PWM, 8-bit
        TCCR1B = (1 << CS10);                           // No prescaler
        OCR1A = 0;  // Initial duty cycle for red
        OCR1B = 0;  // Initial duty cycle for green

        // Configure Timer 2 (8-bit) for PWM on OC2A pin (blue channel)
        TCCR2A = (1 << COM2A1) | (1 << WGM20);  // Fast PWM
        TCCR2B = (1 << CS20);               // No prescaler
        OCR2A = 0;  // Initial duty cycle for blue

        // Set RGB LED pins as outputs (PB1, PB2, PB3)
        DDRB |= (1 << 1) | (1 << 2) | (1 << 3);
}

/**
 * Set RGB LED color
 * @param r Red intensity (0-255)
 * @param g Green intensity (0-255)
 * @param b Blue intensity (0-255)
 */
void set_rgb(uint8_t r, uint8_t g, uint8_t b) {
        OCR1A = r;  // Set red PWM duty cycle
        OCR1B = g;  // Set green PWM duty cycle
        OCR2A = b;  // Set blue PWM duty cycle
}

/**
 * Custom non-blocking delay function
 * Allows the code to respond to inputs during delays
 * @param ms Delay time in milliseconds
 */
void non_blocking_delay(uint16_t ms) {
        for (uint16_t i = 0; i < ms; i++) {
                _delay_ms(1);  // Delay 1ms at a time to check for events

                // Update buzzer for red light penalties or timeout buzzer
```

```
                // This handles the buzzer on/off state during the delay
                if ((button_state && game_state == STATE_COUNTDOWN) ||
timeout_buzzer_active) {
                        PORTB |= (1 << BUZZER_PIN);  // Buzzer ON
                } else {
                        PORTB &= ~(1 << BUZZER_PIN); // Buzzer OFF
                }
        }
}

/**
 * Rainbow color cycle effect for the start screen
 * Smoothly transitions through rainbow colors
 */
void smooth_color_cycle() {
        // Red to purple transition (increase blue)
        for (uint16_t i = 0; i < 256; i++) {
                set_rgb(i, 0, 255-i);
                non_blocking_delay(5);
                if (button_pressed && game_state == STATE_WAITING_TO_START) {
                        return; // Exit if button pressed during start screen
                }
        }

        // Purple to yellow transition (decrease red, increase green)
        for (uint16_t i = 0; i < 256; i++) {
                set_rgb(255-i, i, 0);
                non_blocking_delay(5);
                if (button_pressed && game_state == STATE_WAITING_TO_START) {
                        return;
                }
        }

        // Yellow to teal transition (decrease green, increase blue)
        for (uint16_t i = 0; i < 256; i++) {
                set_rgb(0, 255-i, i);
                non_blocking_delay(5);
                if (button_pressed && game_state == STATE_WAITING_TO_START) {
                        return;
                }
        }

        // Teal to red transition (increase red, decrease blue)
        for (uint16_t i = 0; i < 256; i++) {
                set_rgb(i, 0, 255-i);
                non_blocking_delay(5);
```

```
                if (button_pressed && game_state == STATE_WAITING_TO_START) {
                        return;
                }
        }
}

/**
 * CONVENIENCE FUNCTIONS FOR SETTING COMMON COLORS
 * These make the code more readable when setting colors
 */
// Set RGB LED to red (for countdown/wait state)
void setRed() {
        set_rgb(255, 0, 0);
}

// Set RGB LED to green (for reaction time test state)
void setGreen() {
        set_rgb(0, 255, 0);
}

// Set RGB LED to blue (for results display)
void setBlue() {
        set_rgb(0, 0, 255);
}

// Set RGB LED to orange (for game over state)
void setOrange() {
        set_rgb(255, 165, 0);
}

// Set RGB LED to yellow (for difficulty selection state)
void setYellow() {
        set_rgb(255, 255, 0);
}


/**
 * Configure button interrupt for player input
 * Sets up INT0 to trigger on both press and release
 */
void setup_button_interrupt() {
        // Set PD2 as input with pull-up resistor enabled
        DDRD &= ~(1 << BUTTON_PIN);  // Set as input
        PORTD |= (1 << BUTTON_PIN);  // Enable pull-up resistor

        // Configure INT0 to trigger on ANY logic change (both press and release)
```

```
	// This allows detecting both button press and release events
	EICRA |= (1 << ISC00);	// ISC01=0, ISC00=1: any logical change
	EICRA &= ~(1 << ISC01);

	// Enable INT0 interrupt
	EIMSK |= (1 << INT0);

	// Clear any pending interrupt flag
	EIFR |= (1 << INTF0);
}

/**
 * Generate random delay for red light phase
 * @return Random delay between 1-3 seconds (1000-3000 ms)
 */
uint16_t get_random_delay() {
	return (rand() % 2000) + 1000;  // Random value between 1000-3000ms
}

/**
 * Reset game state for a new game
 * Clears all round data and reaction times
 */
void reset_game() {
	current_round = 0;
	total_reaction_time = 0;
	// Clear all stored reaction times
	for (uint8_t i = 0; i < MAX_ROUNDS; i++) {
		reaction_times[i] = 0;
	}
}

/**
 * Set game difficulty and timeout values
 * @param level Difficulty level (DIFFICULTY_EASY, DIFFICULTY_MEDIUM, or
DIFFICULTY_HARD)
 */
void set_difficulty(uint8_t level) {
	difficulty_level = level;

	// Set green light timeout based on difficulty level
	switch (level) {
		case DIFFICULTY_EASY:
		green_light_timeout = 3000;  // 3 seconds for easy
		break;
		case DIFFICULTY_MEDIUM:
```

```
                green_light_timeout = 1500;  // 1.5 seconds for medium
                break;
                case DIFFICULTY_HARD:
                green_light_timeout = 500;   // 0.5 seconds for hard
                break;
                default:
                green_light_timeout = 1500;  // Default to medium
        }
}

/**
 * Apply timing compensation to reaction time
 * Accounts for system delays in the reaction time measurement
 * @param raw_time The measured reaction time
 * @return Compensated reaction time
 */
uint16_t compensate_timing(uint16_t raw_time) {
        return raw_time + TIMING_COMPENSATION;  // Add compensation value
}

/**
 * MAIN PROGRAM FUNCTION
 * Contains initialization and main game loop
 */
int main(void) {
        // Initialize hardware components
        pwm_init();       // Setup PWM for RGB LED control
        OLED_Init();      // Initialize OLED display
        OLED_Clear();     // Clear the display

        // Configure I/O pins
        DDRB |= (1 << BUZZER_PIN);   // Set PB0 as output for buzzer
        setup_button_interrupt();    // Configure button interrupt on PD2

        // Initialize EEPROM if needed for high scores
        init_eeprom();

        // Seed random number generator
        srand(42);  // Fixed seed for consistent behavior

        // Enable global interrupts to allow button interrupt to function
        sei();

        // Initialize game state
        reset_game();
```

```
/**
 * MAIN GAME LOOP
 * State machine implementation for game flow
 */
while (1) {
        // Switch based on current game state
        switch(game_state) {
                /**
                 * WAITING TO START STATE
                 * Display welcome screen and wait for button press to start
                 * Can go to diffuculty selection from here
                 */
                case STATE_WAITING_TO_START:
                // Display startup screen
                OLED_Clear();
                OLED_SetCursor(0, 0);
                OLED_Printf("Welcome!");
                OLED_SetCursor(1, 1);
                OLED_Printf("Press Button to Start");
                OLED_SetCursor(2, 2);
                OLED_Printf("5-Round Challenge!");

                // Rainbow effect during start screen
                button_pressed = 0;
                while (!button_pressed) {
                        smooth_color_cycle();  // Run rainbow animation until button
pressed
                }

                // Go to difficulty select screen
                game_state = STATE_DIFFICULTY_SELECT;
                button_pressed = 0;   // Reset button flags
                button_released = 0;
                break;


                /**
                 * DIFFICULTY SELECT STATE
                 * Allow player to choose difficulty level
                 * Can go to countdown from here
                 */
                case STATE_DIFFICULTY_SELECT:
                _delay_ms(100);  // Small delay to prevent immediate selection

                // Display difficulty selection screen
                OLED_Clear();
```

```
OLED_SetCursor(0, 0);
OLED_Printf("Choose Difficulty:");
OLED_SetCursor(2, 2);
OLED_Printf("Press to cycle");

// Variables for tracking user selection
uint8_t current_selection = 0;        // Currently selected difficulty
uint8_t selection_confirmed = 0;      // Whether selection is confirmed
uint16_t selection_hold_time = 0;     // How long button has been held

// Display initial difficulty option
OLED_SetCursor(1, 1);
OLED_Printf("> EASY <");

// Set yellow color for selection screen
setYellow();

// Wait for selection and confirmation
while (!selection_confirmed) {
        // If button is RELEASED, cycle through difficulties
        // This is key - only cycle when button is released to avoid
skipping difficulties
        if (button_released) {
                current_selection = (current_selection + 1) % 3;  // Cycle
through 3 options

                // Update display based on current selection
                OLED_SetCursor(1, 1);
                switch (current_selection) {
                        case 0:
                        OLED_Printf("> EASY <  ");
                        break;
                        case 1:
                        OLED_Printf("> MEDIUM <");
                        break;
                        case 2:
                        OLED_Printf("> HARD <  ");
                        break;
                }

                button_released = 0;  // Reset the release flag
        }

        // Check if button is being held to confirm selection
        if (button_state) {
                selection_hold_time += 10;  // Increment hold timer
```

```
                                        // Only show progress if held longer than 300ms (to avoid
flickering during quick presses)
                                        if (selection_hold_time > 300) {
                                                // Show progress at bottom of screen (updates every
100ms)
                                                if (selection_hold_time % 100 == 0) {
                                                        OLED_SetCursor(3, 3);
                                                        OLED_Printf("Hold to confirm:%d%%",
(selection_hold_time - 300) / 17);
                                                }
                                        }

                                        // If held for 2 seconds (2000ms), confirm selection
                                        if (selection_hold_time >= 2000) {
                                                selection_confirmed = 1;
                                        }
                                } else {
                                        selection_hold_time = 0;  // Reset hold timer when button
released

                                        // Clear the progress text when button is released before
threshold
                                        if (selection_hold_time < 300) {
                                                OLED_SetCursor(3, 3);
                                                OLED_Printf("                ");
                                        }
                                }

                                non_blocking_delay(10);  // Small delay for UI responsiveness
                        }

                        // Set game difficulty based on player's selection
                        set_difficulty(current_selection);
                        _delay_ms(135);  // Small delay before showing confirmation

                        // Show confirmation message for selected difficulty
                        OLED_Clear();
                        OLED_SetCursor(0, 0);
                        switch (difficulty_level) {
                                case DIFFICULTY_EASY:
                                OLED_Printf("EASY MODE");
                                OLED_SetCursor(1, 1);
                                OLED_Printf("3 sec to respond");
                                break;
                                case DIFFICULTY_MEDIUM:
```

```
            OLED_Printf("MEDIUM MODE");
            OLED_SetCursor(1, 1);
            OLED_Printf("1.5 sec to respond");
            break;
            case DIFFICULTY_HARD:
            OLED_Printf("HARD MODE");
            OLED_SetCursor(1, 1);
            OLED_Printf("0.5 sec to respond");
            break;
}

// Show "Get Ready!" message before starting game
OLED_SetCursor(3, 3);
OLED_Printf("Get ready!");
non_blocking_delay(2000);  // 2 second countdown

// Start the game with round 1
reset_game();
current_round = 1;
game_state = STATE_COUNTDOWN;
button_pressed = 0;
button_released = 0; // Reset button flags
break;


/**
 * COUNTDOWN STATE (RED LIGHT)
 * Show red light and wait random time before going to green
 * Can go to lose from here and green light
 */
case STATE_COUNTDOWN:
// Display round information
OLED_Clear();
OLED_SetCursor(0, 0);
OLED_Printf("Round %d of %d", current_round, MAX_ROUNDS);
OLED_SetCursor(1, 1);
OLED_Printf("Wait for GREEN light!");

// Set LED to red to indicate "wait" state
setRed();

// Wait random time (1-3 seconds) before switching to green
uint16_t wait_time = get_random_delay();
for (uint16_t i = 0; i < wait_time; i++) {
        non_blocking_delay(1);  // Check for button press each millisecond
```

```
        // If button pressed during red light, game over (false start)
        if (button_pressed) {
                game_state = STATE_LOSE;
                break;
        }
}

// If we successfully completed countdown without button press
if (game_state == STATE_COUNTDOWN) {
        // Transition to green light state
        game_state = STATE_GREEN_LIGHT;
        reaction_time = 0;  // Reset reaction time counter
        button_pressed = 0;  // Clear button flag
}
break;


/**
 * LOSE STATE
 * Player pressed button during red light (false start)
 * Can go back to waiting to start from here
 */
case STATE_LOSE:
// Show "YOU LOSE" message
OLED_Clear();
OLED_SetCursor(0, 0);
OLED_Printf("YOU LOSE!");
OLED_SetCursor(1, 1);
OLED_Printf("Pressed during RED");
OLED_SetCursor(3, 3);
OLED_Printf("Press button");
OLED_SetCursor(4, 4);
OLED_Printf("to try again");

// Set LED to red to indicate failure
setRed();

// Wait for button press to restart
button_pressed = 0;
while (!button_pressed) {
        non_blocking_delay(100);
}

// Reset game and go back to start screen
button_pressed = 0;
button_released = 0; // Reset button flags
```

```
reset_game();
game_state = STATE_WAITING_TO_START;
break;


/**
 * GREEN LIGHT STATE
 * LED is green, player must press button quickly
 * Measures reaction time
 * Can go back to Result or Waiting to start from here
 */
case STATE_GREEN_LIGHT:
// Set LED to green to indicate "go" state
setGreen();
OLED_Clear();
OLED_SetCursor(0, 0);
OLED_Printf("GREEN! Press button!");

// Wait for button press or timeout based on difficulty
while (!button_pressed && reaction_time < green_light_timeout) {
        non_blocking_delay(1);  // 1ms increments for timing accuracy
        reaction_time++;  // Count milliseconds elapsed
}

// Process results based on whether button was pressed in time
if (button_pressed) {
        // Apply timing compensation to reaction time
        uint16_t compensated_time = compensate_timing(reaction_time);

        // Store reaction time (with compensation)
        reaction_times[current_round-1] = compensated_time;
        total_reaction_time += compensated_time;

        // Update reaction_time for display
        reaction_time = compensated_time;

        // Go to result display state
        game_state = STATE_RESULT;
} else {
        // Timeout - player was too slow
        OLED_Clear();
        OLED_SetCursor(0, 0);
        OLED_Printf("Too slow!");
        OLED_SetCursor(1, 1);
        OLED_Printf("You lose!");
```

```
                    // Set LED to red to indicate failure
                    setRed();

                    // Set timeout buzzer flag ON before delay
                    timeout_buzzer_active = 1;

                    // Sound buzzer for timeout
                    PORTB |= (1 << BUZZER_PIN);  // Buzzer ON (redundant but
safe)
                    non_blocking_delay(500);     // Sound buzzer for .5 second

                    // Turn off timeout buzzer flag after use
                    timeout_buzzer_active = 0;
                    PORTB &= ~(1 << BUZZER_PIN); // Buzzer OFF (redundant but
safe)

                    // Additional delay to complete the 2-second pause
                    non_blocking_delay(1500);

                    // Go back to start screen
                    button_pressed = 0;
                    button_released = 0; // Reset button flags
                    reset_game();
                    game_state = STATE_WAITING_TO_START;
                }

                button_pressed = 0;  // Reset button flag
                break;


                /**
                 * RESULT STATE
                 * Show results for current round
                 * Can go to game over and countdown from here
                 */
                case STATE_RESULT:
                // Set blue color for results screen
                setBlue();
                OLED_Clear();

                // Show round number and reaction time
                OLED_SetCursor(0, 0);
                OLED_Printf("Round %d: %d ms", current_round, reaction_time);
                OLED_SetCursor(1, 1);

                // Give feedback based on reaction time
```

```
                    if (reaction_time < 200) {
                        OLED_Printf("Amazing! ");       // Under 200ms - exceptional
                    } else if (reaction_time < 400) {
                        OLED_Printf("Great! ");         // 200-400ms - very good
                    } else if (reaction_time < 600) {
                        OLED_Printf("Good! ");          // 400-600ms - above average
                    } else if (reaction_time < 800) {
                        OLED_Printf("Eh! ");            // 600-800ms - mediocre
                    } else {
                        OLED_Printf("Yea ur bad ");     // 800ms or more - slow reaction
                    }

                    // Wait before moving to next round
                    non_blocking_delay(3000);          // Pause to allow player to see result

                    // Reset button flags to prepare for next state
                    button_pressed = 0;
                    button_released = 0;

                    // Check if all rounds are completed
                    if (current_round >= MAX_ROUNDS) {
                        game_state = STATE_GAME_OVER;   // Move to game over
state
                    } else {
                        current_round++;               // Proceed to next round
                        game_state = STATE_COUNTDOWN;  // Start next round with
countdown
                    }
                    break;


                /**
                 * GAME OVER STATE
                 * Display final average score and top 3 high scores
                 * Wait for player to hold button for 3 seconds to restart
                 */
                case STATE_GAME_OVER:
                    OLED_Clear();                  // Clear OLED for final display
                    setOrange();                   // Set LED to orange for game over

                    // Calculate average reaction time
                    uint16_t average = total_reaction_time / MAX_ROUNDS;

                    // Display the difficulty level completed
                    OLED_SetCursor(0, 0);
                    switch (difficulty_level) {
```

```
        case DIFFICULTY_EASY:
            OLED_Printf("Completed on EASY!");
            break;
        case DIFFICULTY_MEDIUM:
            OLED_Printf("Completed on MEDIUM!");
            break;
        case DIFFICULTY_HARD:
            OLED_Printf("Completed on HARD!");
            break;
}

// Display the average reaction time
OLED_SetCursor(1, 1);
OLED_Printf("Average Time: %d ms", average);

// Check and update top scores
update_top_scores(difficulty_level, average);  // Save if it's a top score
read_top_scores(difficulty_level);             // Load updated top scores

// Display top 3 scores
OLED_SetCursor(2, 2);
OLED_Printf("Top Times:");
OLED_SetCursor(3, 3);
OLED_Printf("1. %d ms", top_scores[difficulty_level][0]);
OLED_SetCursor(4, 4);
OLED_Printf("2. %d ms", top_scores[difficulty_level][1]);
OLED_SetCursor(5, 5);
OLED_Printf("3. %d ms", top_scores[difficulty_level][2]);

// Prompt user to hold button to restart
OLED_SetCursor(6, 6);
OLED_Printf("HOLD 3s to restart");

// Wait for initial button press
button_pressed = 0;
while (!button_pressed) {
    non_blocking_delay(100);       // Check every 100ms
}
button_pressed = 0;

// Wait until button is held continuously for 3 seconds
uint16_t restart_hold_time = 0;
uint8_t showing_progress = 0;
OLED_SetCursor(7, 7);

while (restart_hold_time < 3000) {
```

```
                        if (!(PIND & (1 << BUTTON_PIN))) {  // Button is being held
                                restart_hold_time += 10;

                                // Update progress bar every 300ms
                                if (restart_hold_time % 300 < 10 && !showing_progress) {
                                        showing_progress = 1;
                                        uint8_t progress = restart_hold_time / 300;
                                        OLED_SetCursor(7, 7);

                                        // Draw progress bar
                                        OLED_Printf("[");
                                        for (uint8_t i = 0; i < 10; i++) {
                                                if (i < progress) {
                                                        OLED_Printf("=");  // Filled portion
                                                } else {
                                                        OLED_Printf(" ");  // Empty portion
                                                }
                                        }
                                        OLED_Printf("]");
                                } else if (restart_hold_time % 300 >= 10) {
                                        showing_progress = 0;
                                }
                        } else {
                                // Button released too soon, reset progress
                                restart_hold_time = 0;
                                OLED_SetCursor(7, 7);
                                OLED_Printf("[         ]");  // Clear bar
                        }

                        non_blocking_delay(10);  // Check every 10ms
                }

                // Restart the game after hold is complete
                OLED_SetCursor(7, 7);
                OLED_Printf("Restarting...");
                non_blocking_delay(1000);  // Short delay before reset

                // Reset flags and return to start screen
                button_pressed = 0;
                button_released = 0;
                game_state = STATE_WAITING_TO_START;
                break;
        }
}

return 0;  // Should never reach here
```

```
}
```