THOMPSON RIVERS UNIVERSITY

CENG 3020 – Real Time Systems Design and Analysis

Slot Machine

Isaac Latta (T00239008)

March 25, 2025

Table of Contents

List of Figures

List of Tables

1 Introduction

This report details the design, implementation, and performance analysis of a real-time slot machine game developed on the STM32F4 Discovery board using FreeRTOS. In today's embedded systems, ensuring timely responses and robust functionality is critical; therefore, this project integrates hardware and software components to deliver an interactive, engaging user experience. The design utilizes a modular, event-driven architecture that employs dedicated tasks for polling a push-button, executing LED animations, and managing state transitions via a single-server event queue. Additionally, a pseudo-random number generator—with dual seeding methods for debug and release modes—is used to provide unpredictable outcomes essential to the game’s operation.

The rationale behind the design is to meet stringent real-time constraints while maintaining low CPU utilization and predictable system behavior. By decoupling event generation from processing and clearly segregating task responsibilities, the system achieves a balance between responsiveness and resource efficiency.

Following this introduction, the report is organized as follows: the Design Problem section outlines the project objectives and requirements; the Solution section reviews initial approaches, details the final solution—including features, software architecture, system interfaces, and user interface design—and provides various technical diagrams (feature trees, use case diagrams, activity diagrams, and timing diagrams). Further sections present a requirements traceability matrix, discuss environmental, societal, safety, and economic considerations, and address limitations and potential future work. Detailed results and low-level implementation details are not included here but are discussed in subsequent sections to maintain a clear, high-level overview.

2 Design Problem

This section outlines the central challenge of the project and defines the requirements that guided the system design.

2.1 Problem Definition

The goal is to design and implement a real-time slot machine game for the STM32F4 Discovery board using FreeRTOS. The application must generate unpredictable outcomes using a pseudo-random mechanism, manage multiple concurrent tasks, and respond promptly to user interactions. A button press triggers a spin sequence that decelerates gradually until a specific LED is selected. Depending on whether the LED has been previously “collected,” the game either continues or resets, with visual feedback provided through distinct animations for winning, losing, or progress updates.

2.2 Design Requirements

2.2.1 Functions and Objectives

Functional Requirements (Functions):

- Spin Activation: Initiate a spin sequence when the user presses the button.
- Deceleration Mechanism: Gradually slow the spin until it stops on a designated LED.
- Outcome Determination: Use a pseudo-random process to ensure each game episode is independent and unpredictable.
- Visual Feedback: Display animations and control LED sequences to indicate game states such as win, loss, or collection progress.
- Inter-task Communication: Coordinate operations between game control, LED management, and animation tasks through event queues and callbacks.

Design Objectives:

- Responsive: Ensure timely feedback to user inputs and rapid execution of tasks.
- Robust: Maintain reliable operation under varying conditions with minimal errors.
- Efficient: Optimize resource usage within the limited computational and memory capacities of the STM32F4 platform.
- Engaging: Provide an interactive and captivating user experience.
- Predictable: Achieve consistent task scheduling and execution that meets strict real-time deadlines.

2.2.2 Non-functional requirements and constraints

- Real-Time Responsiveness: The system must adhere to strict timing constraints, responding to inputs and executing tasks within predetermined deadlines.
- Resource Limitations: The design must operate efficiently within the hardware constraints of the STM32F4 Discovery board, ensuring minimal overhead and preventing resource starvation.
- Reliability and Safety: Each gameplay episode should be completely independent to avoid predictable patterns, and the system must be resilient to errors to ensure overall stability.
- Cost-Effectiveness: The implementation should be economical in terms of power consumption and resource utilization, aligning with both performance and safety

requirements.

This detailed articulation of both the functional requirements and the design objectives, alongside the non-functional constraints, provides a clear foundation for the subsequent design and solution exploration presented in later sections of this report.

3 Solution

In this section, we detail the evolution of the design approach, including the technical considerations and scheduling analyses that led to the final solution. I begin by reviewing earlier solutions and their limitations, then present the final, highly optimized design that meets all functional and non-functional requirements.

3.1 Solution 1

The initial implementation combined spin control, LED updating, and animation management in a monolithic task. This approach relied on blocking delays and simple polling, which led to several challenges:

- Poor Responsiveness: The single-task model was unable to provide immediate feedback to user inputs.
- Scheduling Limitations: Blocking delays and static priorities did not account for task jitter or the impact of shared resources.
- Lack of Scalability: With all functionalities combined, the system was not modular and could not easily adapt to varying game conditions or future extensions.

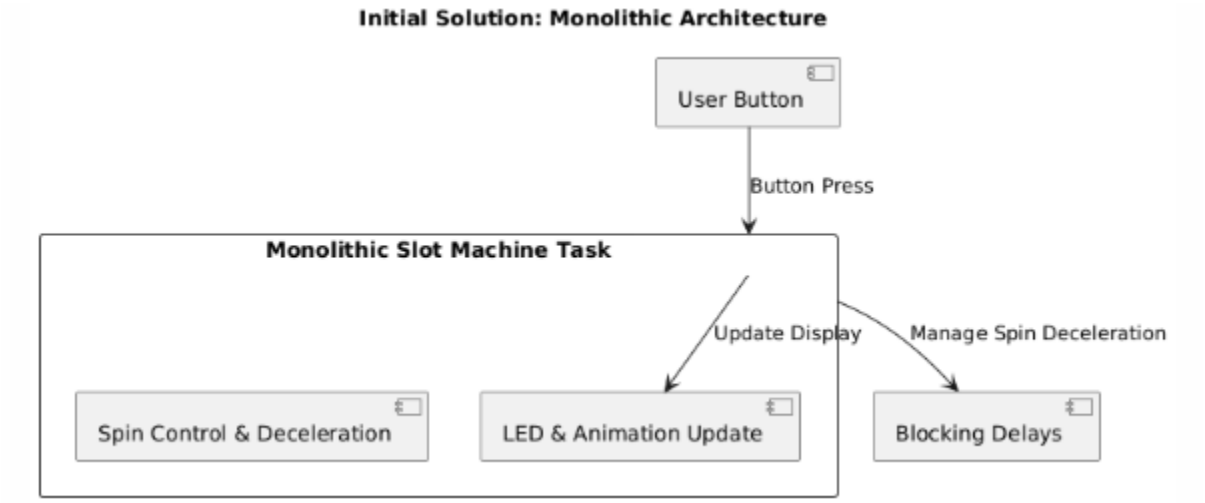

Figure 1: Component diagram for the initial monolithic solution, showing how a single task handles spin control, LED updates, and animations.

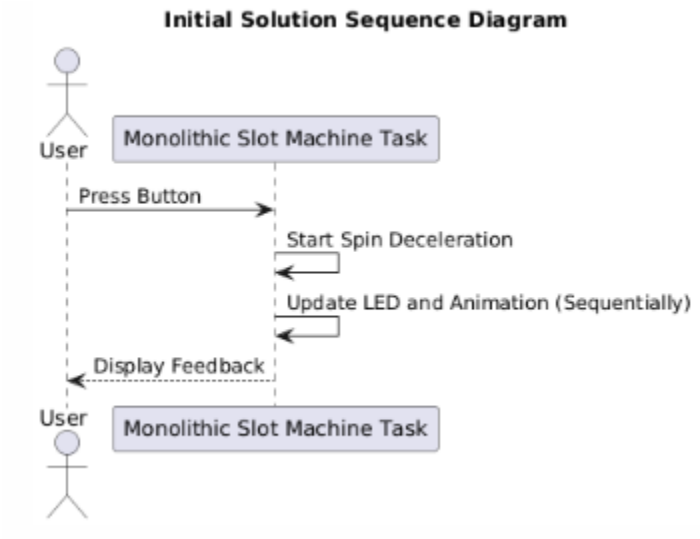

Figure 2: Sequence diagram for the initial solution, where the monolithic task processes the entire spin operation in a single flow.

3.2 Solution 2

The alternative solution decomposed the system into dedicated tasks:

- Spin Task: Handles the deceleration algorithm for the spinning wheel.
- LED Task: Manages LED state updates based on game outcomes.
- Animation Task: Controls visual feedback, using callbacks with void* parameters for flexibility.
- Event Manager: Facilitates inter-task communication via FreeRTOS queues.

This architecture improved responsiveness and modularity but revealed challenges in dealing with shared resources and task jitter. In particular, when multiple tasks required access to the LED control hardware (a non-preemptable resource), priority inversion became a potential issue.

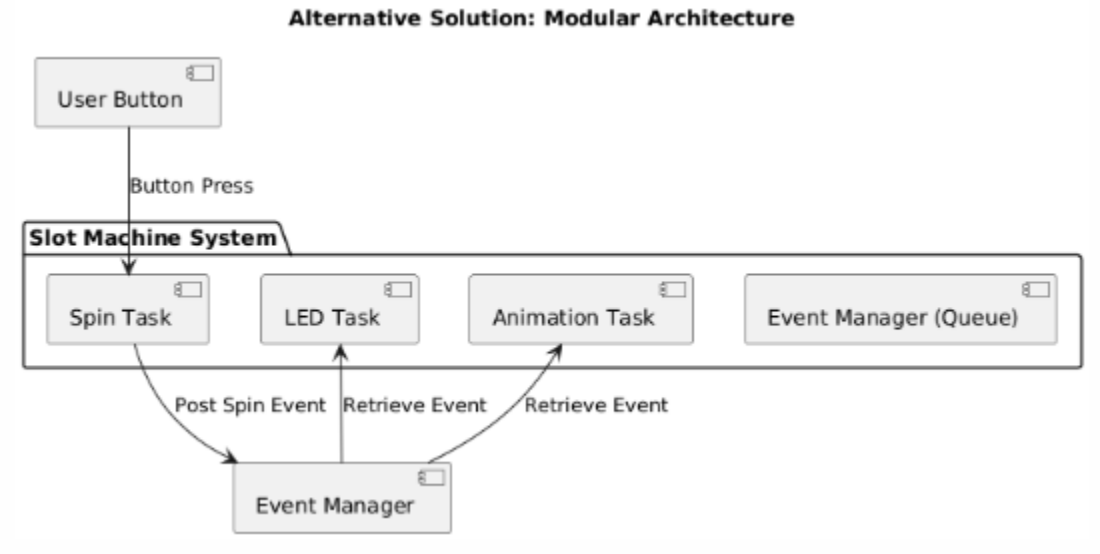

Figure 3: Component diagram for the alternative solution, showing the separate tasks (Spin, LED, and Animation) coordinated via an event manager (queue).

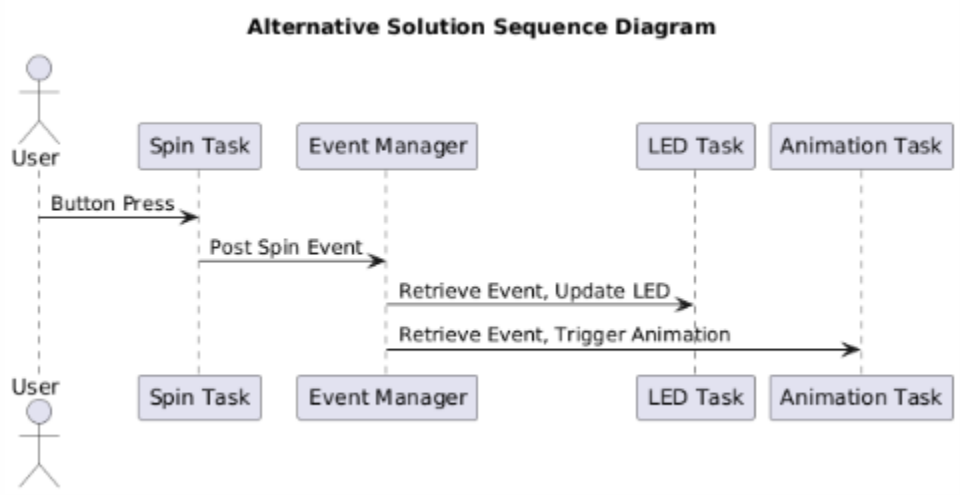

Figure 4: Sequence diagram for the alternative solution, illustrating the separation of concerns between tasks and the role of the event manager in coordinating actions.

3.3 Final Solution

The final solution is implemented using three dedicated tasks that each handle a distinct aspect of the slot machine's functionality:

- **PollButtonTask:** Monitors the hardware button and posts button-press events.
- **AnimateTask:** Executes visual animations (wheel, win, lose, or collected) as requested via the animation queue.
- **StateMachineTask:** Acts as the central controller by processing system events, managing game state transitions (e.g., SPINNING, IDLE, RESET_TO_IDLE), and setting up the appropriate animations.

Because each task has clearly segregated responsibilities, there is no contention for shared resources. Communication is solely via event queues, which minimizes CPU overhead and ensures that each task spends most of its time blocked, resulting in very low overall CPU utilization.

3.3.1 Features

The table below summarizes the key features of the final solution along with their descriptions and the tasks responsible for implementing them.

Feature	Description	Responsible Task	Comments
Button Polling	Continuous monitoring of the push-button to detect user input.	PollButtonTask	Uses a debounce delay (200 ms) to avoid multiple triggers.
Spin Animation	Executes a deceleration animation that simulates a spinning wheel by cycling through LEDs with increasing delays.	AnimateTask (wheelAnimation)	The final LED is chosen randomly by the StateMachineTask.
State Management	Processes events and manages transitions between game states (spin, idle, win, lose).	StateMachineTask	Determines the next animation based on game logic and collected LED status.
Animation Execution	Performs visual feedback through various animations (collected, lose, winning).	AnimateTask	Uses callback functions with void* arguments for flexibility.
Event Queue Communication	Asynchronously passes events between tasks to coordinate actions.	All Tasks	Implements non-blocking and blocking mechanisms to ensure responsiveness.

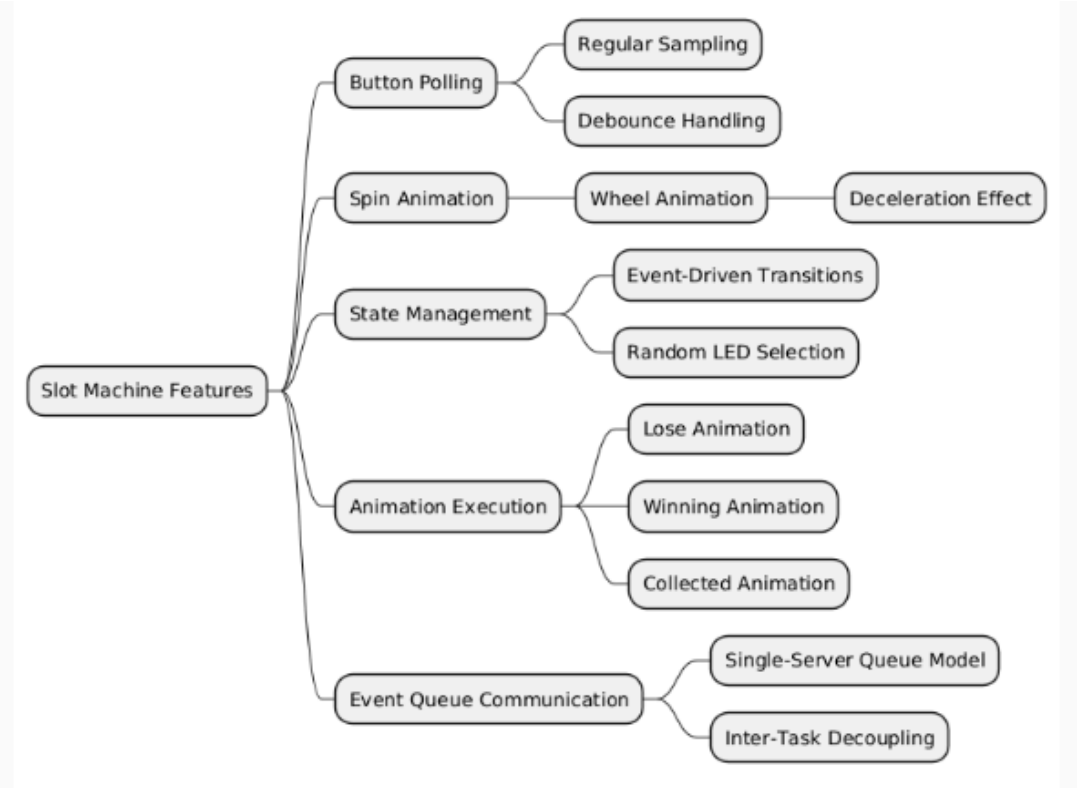

Figure 5: Feature tree showing the hierarchical organization of the key features of the slot machine system.

3.3.2 Use Case Analysis

In this section, a use case diagram is presented that illustrates the primary interactions between the user and the slot machine system. Use cases define the system's functional requirements from the user's perspective. The following are the key use cases:

- **Press Button:** The user initiates the process by physically pressing the push-button. This action is detected by the PollButtonTask, which sends an EVT_BUTTON_PRESS event.

- **Initiate Spin:** Upon receiving the button press event, the StateMachineTask transitions the system to the SPINNING state and calls the spin() function to randomly select a winning LED.

- **Display Wheel Animation:** The AnimateTask executes the wheelAnimation function to visually simulate the spinning of the wheel. This animation uses a deceleration effect with increasing delays between LED transitions.

- **Process Outcome:** After the wheel animation completes, the StateMachineTask evaluates the outcome by comparing the selected LED with the current collected mask, thereby determining if the result is a win, a loss, or an intermediate collected state.

- **Display Feedback (Win/Lose/Collected):** Based on the outcome, the appropriate

animation (winning, losing, or collected) is displayed to provide immediate visual feedback to the user.

The following use case diagram visually represents these interactions:

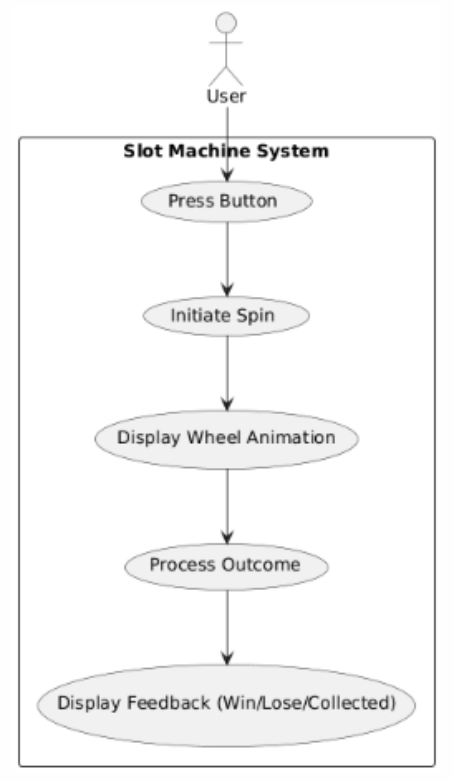

Figure 6: Use case diagram illustrating the primary interactions between the user and the slot machine system.

3.3.3 Activity Diagram

The following activity diagram provides a high-level overview of the slot machine's workflow. It begins with the user pressing the button and shows how each decision point (e.g., whether the LED is already collected) and action (e.g., spinning, updating the collected mask, triggering animations) leads the system from one step to the next. This diagram complements the state machine by illustrating how user events, animations, and internal logic form a coherent process flow.

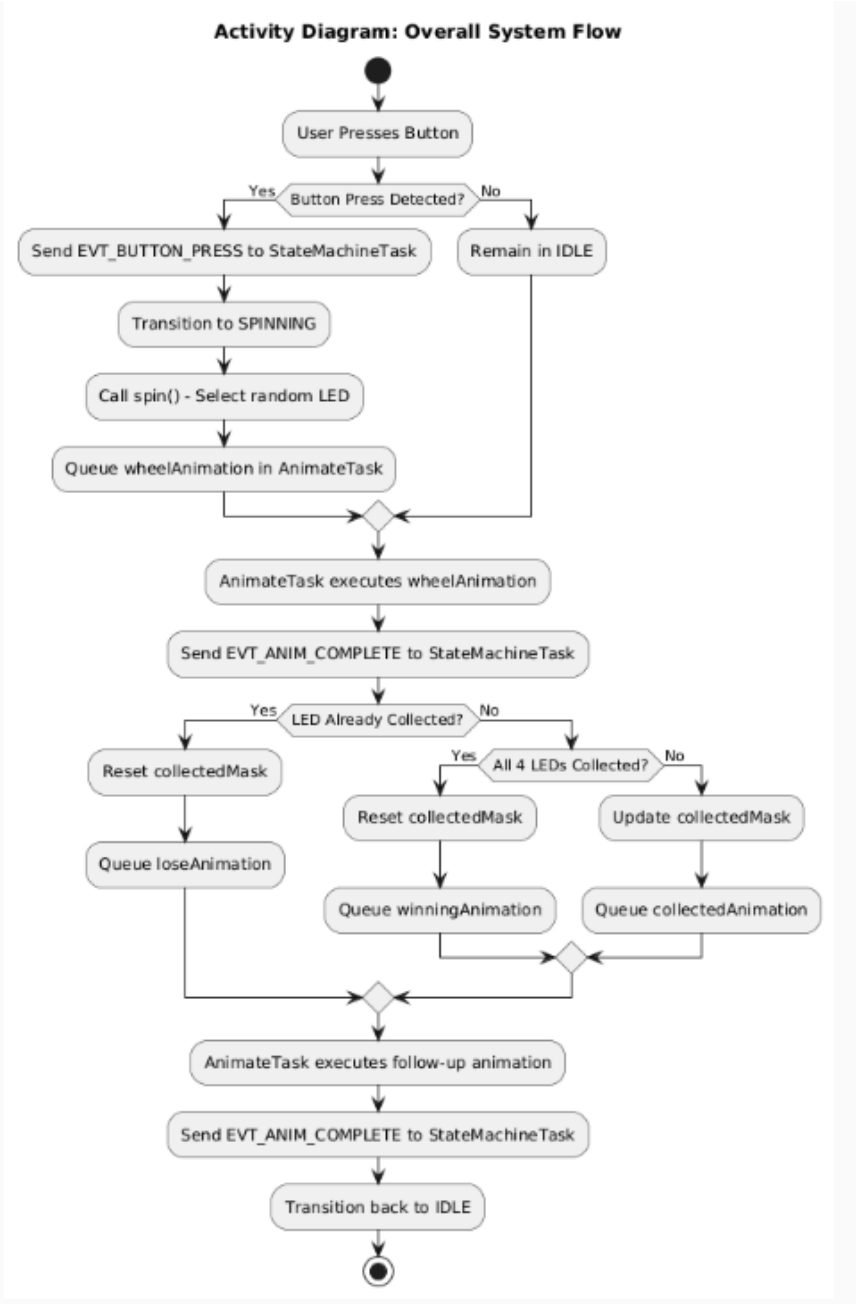

Figure 7: Activity diagram showing the primary workflow from button press to spin, outcome determination, animation execution, and returning to the idle state.

3.3.4 Architecture

The overall architecture is built around three main tasks communicating via a central event queue. The tasks have clearly defined roles, and their interaction ensures that the system responds rapidly to user input while maintaining strict real-time performance.

The design leverages a single-server queue model for event processing. In this model, the central event queue acts as the single server that serially handles all incoming events from multiple producer tasks (e.g., PollButtonTask and AnimateTask). The StateMachineTask serves as the dedicated consumer (or “server”) that retrieves and processes events one at a time, ensuring that state transitions and subsequent actions

are handled sequentially. This approach decouples event generation from event consumption, simplifies synchronization, and prevents resource contention.

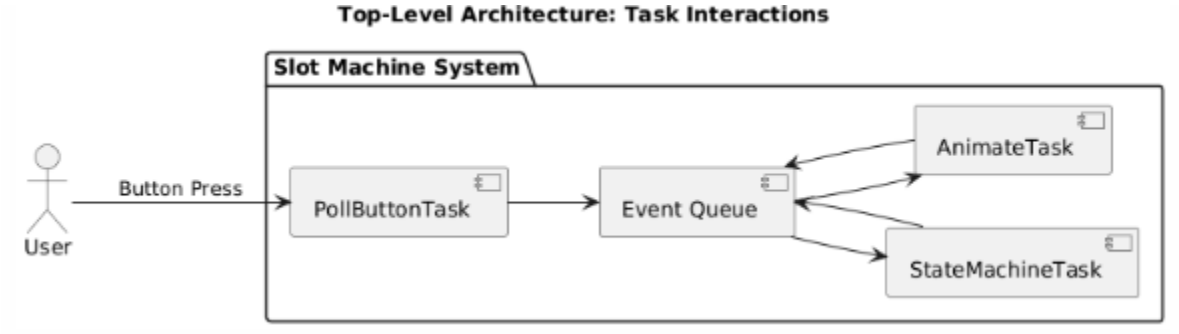

Figure 8: Top-level component diagram showing the three main tasks and their communication via a central event queue.

3.3.4.1 StateMachineTask Internal Structure

The StateMachineTask is responsible for overall game control. Internally, it comprises two main subcomponents:

- Spin Setup:
  - Function: spin()
  - Role: Randomly selects a winning LED (0–3) and prepares the animation (typically, the wheelAnimation) by setting up the necessary arguments.
- State Transition Logic:
  - Function: setNextAnimation()
  - Role: Based on the current game state and the “collected” LED mask, it determines which animation to execute next (collected, lose, or winning).

The sub-components interact by processing events from the event queue and then configuring the next animation to be posted back to the animation queue.

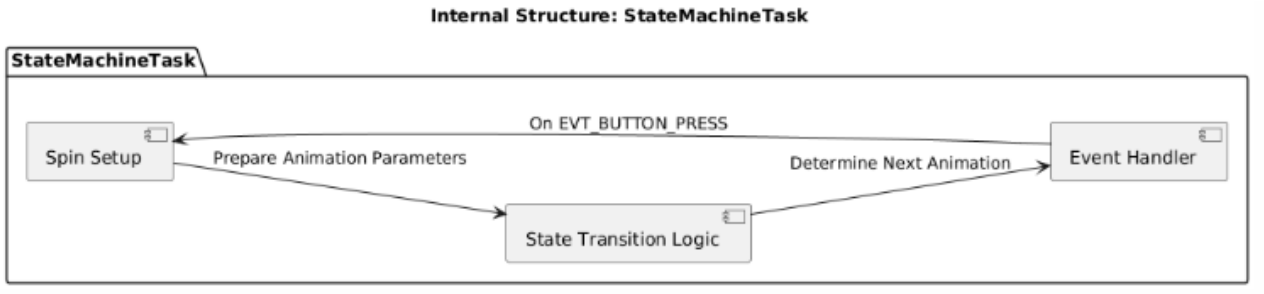

Figure 9: Internal component diagram for StateMachineTask showing the interaction between the spin setup and state transition logic sub-components.

3.3.4.2 AnimateTask Internal Structure

The AnimateTask manages the execution of animations. It primarily:

- Receives Animation Requests: Waits on the animation queue for an Animation_t

structure.
- Executes Animation Functions: Calls the specified animation function (e.g., wheelAnimation, loseAnimation, etc.) using the provided arguments.
- Signals Completion: After finishing an animation, sends an EVT_ANIM_COMPLETE event back to the event queue.

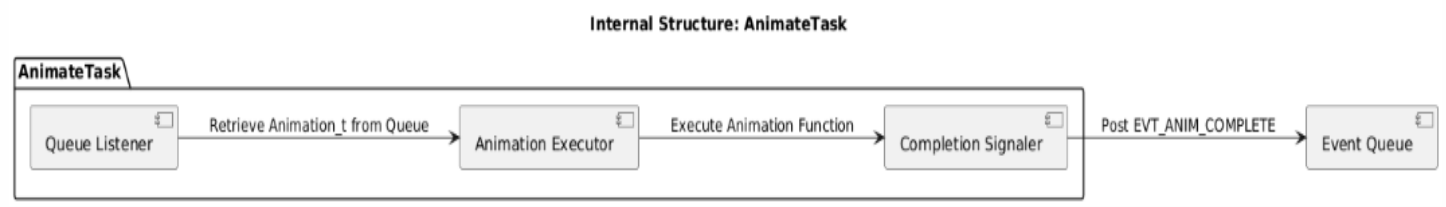

Figure 10: Internal component diagram for AnimateTask illustrating how it retrieves, executes, and signals the completion of animation events.

3.3.5 CPU Utilization

Because each FreeRTOS task in this system spends most of its time blocked (waiting on events or delays), overall CPU utilization is minimal. Below is an approximate calculation based on the observed behavior and typical overhead for each task. These estimates assume a 168 MHz STM32F4 processor, where simple operations (e.g., HAL_GPIO_ReadPin(), queue sends, and short loops) complete in a fraction of a millisecond.

PollButtonTask

- Behavior: Wakes every 50 ms to check the button state. If pressed, it sends an event to the queue and applies a 200 ms debounce delay.
- Reasoning:
  - Checking the button (HAL_GPIO_ReadPin()) plus a queue send typically takes well under 0.1 ms.
  - Thus, each 50 ms cycle consumes ~0.1 ms of active CPU time in the worst case.

- Utilization:

$$U_{PB} = \frac{0.1\,\text{ms}}{50\,\text{ms}} = 0.002 = 0.2\%$$

AnimateTask

- Behavior: Blocks on xQueueReceive(xAnimationQueue, …). When an animation request arrives, it executes the specified animation (e.g., wheelAnimation), which itself mostly waits in vTaskDelay() calls. Afterward, it sends an EVT_ANIM_COMPLETE event and returns to blocking.
- Reasoning:
  - During idle cycles (every 50 ms check), retrieving from the queue and

minimal housekeeping might take ~0.05 ms.
    - A spin animation, for example, may run for ~6 seconds, but the CPU is largely blocked in vTaskDelay() for each LED change. Actual GPIO operations (like HAL_GPIO_WritePin()) are brief.
    - We approximate ~5 ms of active CPU time to handle the entire spin animation sequence (mostly overhead from function calls and minimal loops).
- Idle Utilization:

$$U_{\text{AMT\_idle}} = \frac{0.05\,\text{ms}}{50\,\text{ms}} = 0.001 = 0.1\%$$

- Animation Overhead (spin scenario):

If a spin animation (5 ms CPU time) is triggered once every 6 seconds (immediate re-spin),

$$U_{\text{AMT\_spin}} = \frac{5\,\text{ms}}{6000\,\text{ms}} \approx 0.083\%$$

StateMachineTask

- Behavior: Blocks on xQueueReceive(xEventQueue, …). When it receives an event (e.g., button press or animation complete), it transitions states, calls spin() or setNextAnimation(), and queues any follow-up animations.

- Reasoning:

    - Processing each event typically involves quick checks (e.g., if a collected LED mask is set) and possibly one queue send. This overhead might be ~0.1 ms per event.

    - If the user initiates a spin roughly once every few seconds, the total overhead is extremely small.

- Utilization (assuming 1 spin event every 6 seconds):

$$U_{SM} = \frac{0.1\,\text{ms}}{6000\,\text{ms}} \approx 0.0017\%$$

Overall Utilization

Summing these contributions yields an approximate total CPU utilization:

$$U_{\text{total}} = U_{PB} + U_{AM} + U_{SM} \approx 0.2\% + 0.18\% + 0.002\% \approx 0.38\%$$

Even if the user presses the button more frequently or triggers multiple animations, the overhead remains well below 1–2%. Since typical real-time scheduling bounds are much higher, the system comfortably meets its deadlines with ample CPU headroom.

3.3.6 Randomness and Winning Odds

The system uses a pseudo-random number generator (PRNG) to select a winning LED for each spin. This is achieved by seeding the PRNG using one of two methods:

- **Debug Mode:**

  In debug mode, the seed is set using the current tick count via xTaskGetTickCount(). This approach guarantees consistent behavior across runs, which is valuable for testing and demonstration purposes.

- **Release Mode:**

  In release mode, the seed is obtained from the hardware Random Number Generator (RNG) peripheral. The RNG is initialized through MX_RNG_Init(), and the seed is retrieved using HAL_RNG_GenerateRandomNumber(). The obtained seed is then used to initialize the PRNG with srand(), ensuring non-deterministic and truly random outcomes during normal operation.

The DEBUG macro is used to determine which seeding approach is used. This dual-mode approach ensures that in development, tests and demos produce repeatable results, while in production, the randomness is more robust due to the hardware RNG.

In the game logic, winning is achieved when the player collects all four unique LEDs. The probability of a win is determined by the likelihood that each successive spin selects an LED that has not already been collected. Assuming the following:

- First Spin: Any LED can be chosen (Probability = 1).
- Second Spin: The chance of selecting a new LED (given one LED is already collected) is 3/4 (or 0.75).
- Third Spin: The probability of choosing one of the remaining two new LEDs is 2/4 (or 0.50).
- Fourth Spin: The probability of picking the last remaining LED is ¼ (or 0.25).

The overall probability of winning (i.e., successfully collecting all four unique LEDs in a row) is:

$$P(\text{win}) = 1 \times 0.75 \times 0.50 \times 0.25 = 0.09375$$

This calculation indicates that the odds of winning are approximately 9.375%.

3.3.7 The System Interfaces

In this design, the system is driven by two categories of events:

Temporal (Time-Triggered) Events

- Periodic Button Polling: The PollButtonTask checks the button state every 50 ms. This periodic check is considered a time-triggered event, ensuring regular sampling of the button input.
- Periodic Task Delays: Both the AnimateTask and PollButtonTask include short fixed delays (vTaskDelay) within their loops (e.g., 50 ms for animation idle, 200 ms for debouncing). These periodic waits effectively partition CPU usage and maintain a predictable schedule for each task.

Signal (Event-Driven) Events

- Button Press (External Hardware Signal): When the button is detected as pressed, the PollButtonTask sends an EVT_BUTTON_PRESS to the system event queue. This is an external signal event originating from the user.
- Animation Completion: After an animation finishes (e.g., the wheel spin or a losing/winning sequence), the AnimateTask sends an EVT_ANIM_COMPLETE event to the system event queue, signaling that the animation is done and that the next state transition can proceed.
- State Machine Transitions: The StateMachineTask processes incoming events (EVT_BUTTON_PRESS, EVT_ANIM_COMPLETE, etc.) and responds by changing internal states (e.g., IDLE, SPINNING, RESET_TO_IDLE), as well as posting new animation requests to the animation queue when necessary.

3.3.7.1 Event Types and System Responses

The code defines several EventType_t values that represent the core signals in this system. Below is a brief mapping of each event type to its expected system response:

- EVT_BUTTON_PRESS
  - Origin: PollButtonTask
  - Response: Transition to SPINNING state; invoke spin() to select a random LED and queue a wheelAnimation.
- EVT_ANIM_COMPLETE
  - Origin: AnimateTask (sent after finishing any animation)
  - Response: StateMachineTask calls setNextAnimation() to decide whether to show a losing, winning, or collected animation—or return to the IDLE state.
- EVT_ANY
  - Used internally by StateMachineTask to filter out events that do not match the expected type during certain states.
  - Response: If the event does not match the state's expected type, it is ignored.

By decoupling tasks through a centralized event queue, the design ensures each event is handled predictably under real-time constraints, with minimal blocking and no need for complex resource sharing.

In the slot machine system, the state machine operates through three primary states:

IDLE, SPINNING, and RESET_TO_IDLE. When the system is in the IDLE state, a button press (EVT_BUTTON_PRESS) triggers a transition to SPINNING, during which the system randomly selects a winning LED and executes a wheel animation. Once the wheel animation completes (signaled by EVT_ANIM_COMPLETE), the state transitions to RESET_TO_IDLE, where the system evaluates whether the selected LED has already been collected, subsequently triggering the appropriate follow-up animation (lose, collected, or winning) before finally returning to the IDLE state.

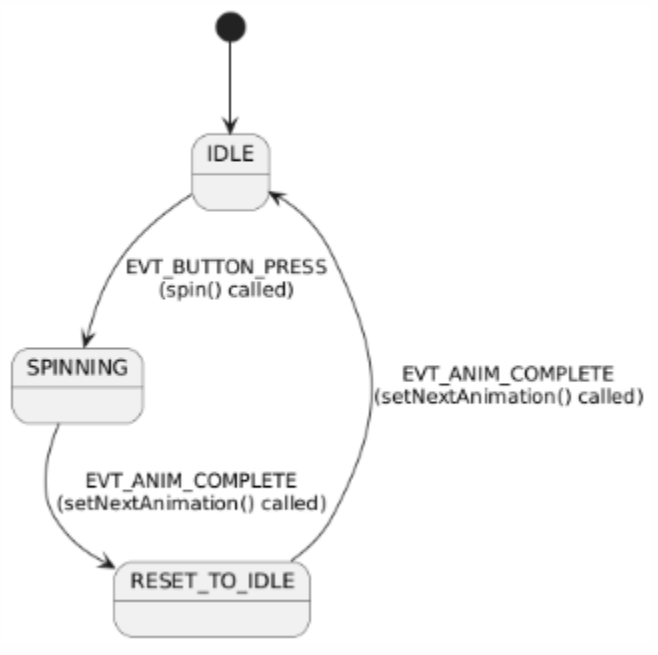

Figure 11: State graph illustrating the core transitions of the slot machine system's state machine.

3.3.8 The User Interface Design

The user interface is composed of a physical push-button, an LED array, and animated feedback sequences. This interface aims to provide immediate, intuitive responses to player actions.

3.3.8.1 Components

- Push-Button
  - Hardware Input: The user's sole direct control mechanism.
  - Software Task: PollButtonTask regularly samples the button state, detecting transitions from not-pressed to pressed.
  - Business Event: Triggers EVT_BUTTON_PRESS in the system event queue, initiating a spin sequence.
- LED Array
  - Hardware Output: Consists of four LEDs (plus an optional red LED for certain animations).
  - Software Control: Each animation function (e.g., wheelAnimation, loseAnimation) manipulates the LEDs to reflect the current game state.
  - Feedback Purpose: Conveys outcomes (spinning, winning, losing, or collected progress) via light patterns.

- Animations
    - Implementation: Encapsulated as function pointers (animation) with optional arguments (args) in the Animation_t structure.
    - Possible Animations:
        - Wheel Animation (wheelAnimation): Simulates the spinning of a wheel by cycling through LEDs with increasing delays.
        - Lose Animation (loseAnimation): Flashes the red LED, followed by a brief display of all LEDs.
        - Winning Animation (winningAnimation): Rapidly cycles through all LEDs in a celebratory pattern.
        - Collected Animation (collectedAnimation): Toggles the LEDs corresponding to collected bits in a quick flashing sequence.

3.3.8.2 Business Events and Responses

Below is a concise mapping of key “business events” in the system to their user-facing responses:

- Spin Request (Button Press):
    - Response: StateMachineTask calls spin(), chooses a random LED, and queues the wheel animation. The user sees a spinning pattern on the LED array.
- Spin Completion (Wheel Animation Ends):
    - Response: StateMachineTask checks whether the chosen LED was previously collected. If yes, triggers a lose animation; if no, sets the LED as collected. If all four LEDs become collected, triggers a winning animation.
- Lose Animation Completion:
    - Response: The system resets the “collected” mask and returns to the IDLE state.
- Win Animation Completion:
    - Response: Similarly resets the “collected” mask and returns to the IDLE state.

The following sequence diagram illustrates how the system responds when a user initiates a spin. It highlights both the signal events (button press, animation completion) and the user interface feedback (LED animations).

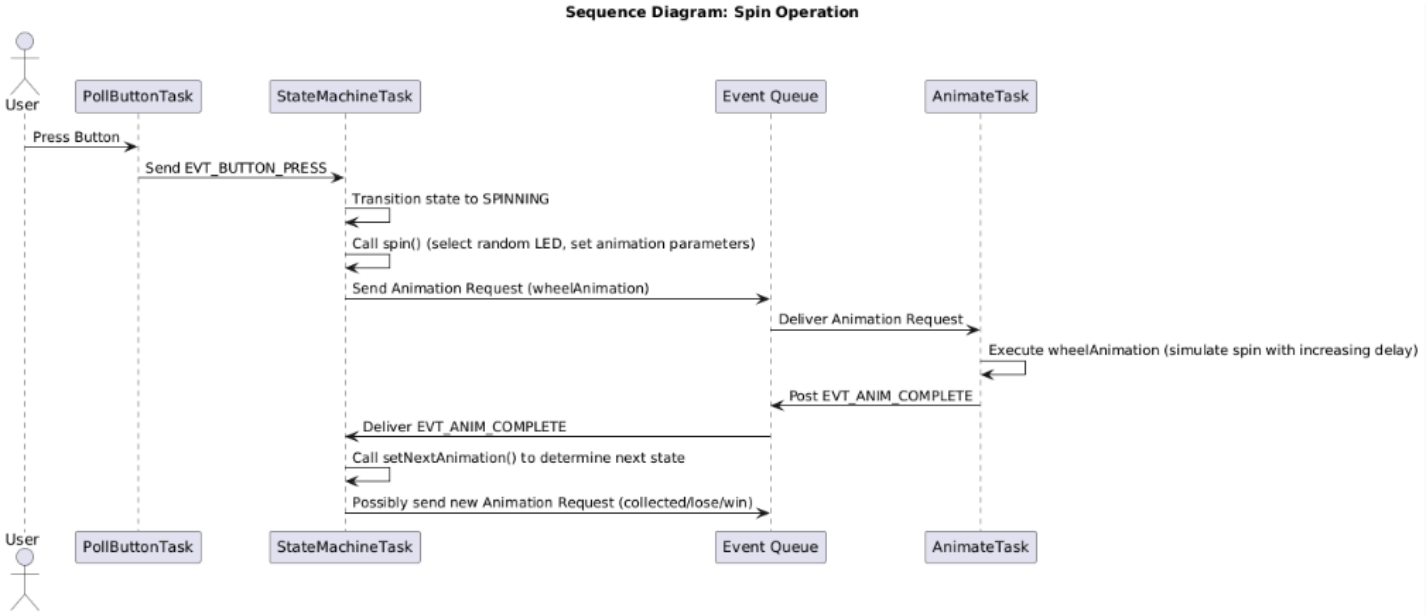

Figure 12: Sequence diagram illustrating the user pressing the button (signal event) and the system's subsequent responses, including state changes, animation execution, and LED update

3.3.9 Requirements Traceability Matrix

Req. ID	Requirement	Design Component	Code Component	Testing Scenario
R1	The system shall initiate a spin sequence upon button press.	PollButtonTask	PollButtonTask() function	Button Press Test: Press the button and verify that an EVT_BUTTON_PRESS event is sent, leading to a new spin in StateMachineTask.
R2	The system shall randomly select a winning LED and display a “spinning” animation.	StateMachineTask	spin() and wheelAnimation()	Spin Animation Test: Upon receiving EVT_BUTTON_PRESS, ensure a random LED is chosen. Observe the wheel animation cycling through LEDs with increasing delay.
R3	The system shall maintain a collected mask and display different animations based on previously collected LEDs.	StateMachineTask	setNextAnimation()	Collected/Winner/Lose Test: Manually press the button until all LEDs are collected. Confirm correct animation (collected/lose/winning) is triggered.
R4	The system shall send an event to signal animation completion.	AnimateTask	AnimateTask() function	Animation Completion Test: Trigger a spin animation, verify EVT_ANIM_COMPLETE is posted upon finishing, and confirm StateMachineTask receives it.
R5	The system shall be responsive to user input, with no noticeable lag.	Event Queues	xQueueSend / xQueueReceive	Responsiveness Test: Repeatedly press the button at short intervals. Confirm that each press is processed without significant delay or missed inputs.

R6	The system shall operate without resource contention or deadlocks.	Task Segregation	N/A (No shared resource needed)	Concurrency Test: Run the application for an extended period, repeatedly pressing the button. Confirm no hangs or unexpected blocking occurs.
R7	The system shall display a lose animation if the selected LED is already collected, and a winning animation if all LEDs are collected.	Animation Mechanisms	loseAnimation(), winningAnimation()	Game Logic Test: Press the button repeatedly, ensuring the lose or winning animation occurs under correct conditions (LED collected vs. all collected).
R8	The system shall remain within real-time constraints (low CPU utilization).	Task Scheduling	vTaskDelay() usage, FreeRTOS	Observe behavior under frequent button presses. Confirm the CPU remains responsive.

3.3.10 Environmental, Societal, Safety, and Economic Considerations

In developing the real-time slot machine system, several potential impacts were examined and benefits in multiple domains. While the project is primarily educational, it adheres to best practices that support broader considerations.

3.3.10.1 Environmental considerations

- Low Power Consumption:
  By relying on FreeRTOS's blocking queues and minimal CPU usage in idle states, the system conserves energy. Tasks that are not actively performing work remain blocked, reducing unnecessary power draw.

- Scalability to Green Solutions:
  This modular architecture can be scaled to run on low-power STM32 boards or integrated with energy-efficient power supplies, minimizing the carbon footprint for any final product deployment.

3.3.10.2 Societal considerations

- Enhanced User Engagement:
  The game-like interface with spinning LEDs and feedback animations encourages interactive learning about real-time systems and embedded programming concepts.
- Educational Value:
  The design demonstrates foundational real-time concepts (scheduling, event-driven tasks) that can be repurposed in other safety-critical or consumer electronics, benefiting society by training engineers in robust, real-time development.

- Responsible Gaming Context:
  Although it's a slot machine simulator, it does not involve real money. This ensures it is a safe tool for demonstration or entertainment without encouraging gambling behaviors.

3.3.10.3 Safety considerations

- Robust State Management:
  The finite-state machine design avoids undefined or contradictory states, reducing the likelihood of unpredictable behavior or system crashes.
- Non-Blocking Communication:
  Using event queues and segregated tasks prevents deadlocks or priority inversions, contributing to a safe and reliable run-time environment.
- Hardware Debounce:
  The 200 ms debounce for the button not only improves user experience but also prevents spurious events that might disrupt the system.

3.3.10.4 Economic considerations

- Cost-Effective Hardware:
  The STM32F4 Discovery board provides a robust development environment at a relatively low cost, making this approach accessible for educational or low-volume product development.
- Reusable Modules:
  By segregating tasks and using a clean event-driven design, many software components (e.g., the animation system or state machine logic) can be reused in other projects, reducing development time and cost.
- Open-Source RTOS:
  FreeRTOS is open-source, minimizing software licensing fees and encouraging collaborative improvements.

3.3.11 Limitations

Despite its strengths, the current design does have certain limitations:

1. Limited Scope of Animations:
   The system only provides a fixed set of animations (wheel, lose, winning, collected). Adding more complex or customizable animations may require additional memory and CPU resources.
2. Single-Button Input:
   The user interface is restricted to one button. Expanding to multiple buttons or input devices would necessitate extra hardware and additional tasks or interrupts.
3. No Persistent Storage:
   The system does not save progress (collected LEDs) across power cycles. Implementing non-volatile memory or SD card storage would require further design.
4. Reliance on vTaskDelay for Timing:
   Although simple, using vTaskDelay introduces reliance on FreeRTOS scheduling ticks. If finer-grained or more precise timing were needed, hardware timers or

interrupt-driven scheduling might be necessary.

5. Basic Randomness:
   The random LED selection is sufficient for demonstration but may not meet rigorous statistical requirements for true randomness in commercial gaming systems.

These constraints do not detract from the educational or illustrative value of the design, but they highlight areas where additional research, hardware, or software complexity might be required for broader or more demanding applications.

4 Teamwork

Although this project was completed individually, the following “meeting” structure captures how tasks were planned, managed, and completed over time.

4.1 Meeting 1

Date: March 10, 2025

Agenda: High-Level Design of Slot Machine

Architecture

Team Member	Previous Task	Completion State	Next Task
Isaac	N/A	N/A	- Define system concept - Outline code structure

Notes:

- Determined project scope and key features (spinning LED wheel, collected/win/lose states).
- Decided on three FreeRTOS tasks: PollButtonTask, AnimateTask, and StateMachineTask.

4.2 Meeting 2

Date: March 11, 2025

Agenda: Code Implementation and Testing

Team Member	Previous Task	Completion State	Next Task
Isaac	- Define system concept - Outline code structure	100%	- Complete Code Implementation - Complete System Testing

Notes:

- Started coding PollButtonTask for button input and AnimateTask for LED animations.
- Set up queues for event-driven communication.
- Integrated the state machine logic (StateMachineTask) with button and animation tasks.
- Performed hardware testing on the STM32F4 Discovery board.

4.3 Meeting 3

Date: March 24, 2025

Agenda: High-Level Design of Slot Machine

Architecture

Team Member	Previous Task	Completion State	Next Task
Isaac	- Complete Code Implementation - Complete System Testing	100%	- Finalize project report

Notes:

- Completed all required documentation, including diagrams and scheduling analysis.

5 Conclusion and Future Work

In this project, a real-time slot machine application was successfully designed and implemented on the STM32F4 Discovery board using FreeRTOS. The system meets its primary objectives by offering responsive button polling, dynamic LED animations, and a clear state machine to manage gameplay events (e.g., spinning, winning, losing). Through the use of event queues and carefully segregated tasks, the design adheres to real-time constraints while avoiding complex resource-sharing issues. Periodic and event-driven interactions are coordinated efficiently, resulting in a robust, low-overhead architecture.

Despite its effectiveness, a few limitations remain—such as limited animations, a single-button interface, and no persistent data storage. These constraints, however, do not diminish the educational value or the illustrative power of the final solution, which demonstrates foundational real-time operating system principles and embedded systems programming techniques.

Future work would include:

1. Expanded User Interface:
   Incorporate multiple buttons or alternative input devices to broaden gameplay options and test more complex scheduling scenarios.

2. Persistent Data Storage:
   Introduce non-volatile memory (e.g., EEPROM or SD card) to save collected LED progress or maintain game statistics across power cycles.

3. Additional Animations and Effects:
   Create more varied visual feedback—such as color-cycling LEDs, multi-pattern transitions, or user-selectable themes—to enrich the gaming experience.

4. Advanced Scheduling Techniques:
   Experiment with alternative or more advanced scheduling methods (e.g., dynamic priorities, Rate Monotonic Analysis for multiple tasks) to explore the scalability and responsiveness of the system under heavier loads.

6 References

Appendix