PinballEErs Final Report

EE Senior Design 2025

Tobin Mosley Bradshaw, Gavin Paul Carr, Allison Elizabeth Fleming, Mary Rose Nelligan, & Clare Elizabeth Nickerson

May 7th, 2025

Table of Contents

1 Introduction

The widespread reliance on screens has created issues in modern life such as excessive scrolling, reduced social interactions, and increased feelings of isolation. This challenge is especially prevalent for stressed students like those in the Notre Dame electrical engineering curriculum. The rigorous curriculum can lead to heightened stress levels and produce a need for effective outlets.

Our EE-themed pinball machine addresses these problems by encouraging users to take a break from their electronic devices and interact with others in a shared, lighthearted environment. It serves as both a stress relief and community-building tool. Although screens and technology are aspects of our project, the focus for the user is the movement of the ball and being present with the people around them.

The pinball machine resonates with students especially by incorporating elements inspired by the electrical engineering curriculum at Notre Dame. For instance, the targets both have an “E” on them, together forming “EE,” an abbreviation for the major. The ramp has the words “Patrick Mahomes ran a 1st down with a sprained ankle, you can make it up this ramp,” which is a reference to Professor Huang’s iconic quote from Signals and Systems. The game’s thematic design fosters a sense of community and camaraderie among students, creating an appreciation for the shared experiences.

Gameplay is enhanced by multimedia feedback. LEDs, display screens, and sound effects add excitement and hilarity to the game by giving users something to socialize over while they play. The inclusion of the high score and the personal scoring system brings out the player’s competitive nature and engages their critical thinking and strategic skills, offering mental stimulation in a manner that goes beyond the passive consumption of social media. Overall, the flashy nature of the machine and the competition involved engages users and allows the machine to be a place for friends to play with each other and enjoy each other’s company.

The pinball machine incorporates several moving and interactive components that emulate the difficulty and excitement of the original game. A simple spring mechanism launches the ball into the gameplay area. A mechanical flipper mechanism allows the player to control the flippers by adjusting their angle to send the ball shooting forward. With regards to obstacles and targets, there are two stationary targets, one moving target, one rotating paddle, and one ramp. The stationary targets each consist of a microswitch with a 3D-printed target on top, as well as a back boundary that prevents false triggers. The moving target operates with a bipolar stepper motor connected to a gear which rotates the circular ring back and forth. There are two metal contacts at the bottom of the ring such that the metal ball pulls a signal high when it goes over. The

rotating paddle operates through a bipolar stepper motor with a two pronged attachment on top. The 3D-printed ramp contains IR beam break sensors at the top that detect when the ball passes through. When the player scores points, the scoreboard updates, images display on the OLED, audio clips play, and lights flash. This provides an engaging and fun user experience.

In terms of software logic, the machine operated in three separate modes: Idle Mode, Gameplay Mode, and Editing Mode:

1. **Idle Mode:** The game awaits a new player, displaying high scores and user initials on the OLED.
2. **Gameplay Mode:** A player launches the ball, allowing for score tracking and all effects as the game demands.
3. **Editing Mode:** If the user achieves a top three score, the machine prompts for initials using two buttons, one to scroll through the letters and one to confirm the user's choice. If a new game begins before initials are complete, the score is saved under “XXX.”

Transitions between these three modes are automated via the IR sensor located by the launcher. While the ball is between the IR beam break sensors, resting against the launcher, the machine is in Idle Mode. When the user loads the ball into the launcher runway by pulling back the spring loaded launcher, the IR beam is restored and the machine enters Gameplay Mode. When the game is over and the ball returns between the beam, the machine enters Editing Mode if the user achieved one of the three high scores.

The microcontroller has enough extra pins to connect to each of the obstacle’s buttons or laser sensors as well as the IR-beam laser sensor in the holding area. Furthermore, it has two SPI outputs: one for the seven segment display and one for the OLED. For power, the microcontroller runs on 3.3V which is stepped down from a 120V wall outlet. The seven segment display and the OLED are powered by 5V – also stepped down from the wall outlet. The LED strip is powered by 12V, stepped down from the wall outlet as well.

The physical enclosure was constructed using quarter inch acrylic sheets, 3D-printed ABS plastic components, and acrylic-specific Weld-On adhesive. As for physical dimensions, the gameplay surface is 12in x 22in, set at a 6.5° angle to ensure optimal ball dynamics and consistency with traditional pinball designs. The total length of the machine is 12.5in x 25.5in. The water-jet cutter at the Engineering Innovation Hub cut the acrylic into the properly sized pieces. The laser cutter removed the smaller, more

precise pieces, such as those for the components on the gameplay area as well as space for wires to go down discreetly. The front casing of the pinball machine was 3D-printed in two pieces and later joined. This is due to size constraints of the 3D-printer at the Engineering Innovation Hub as well as tools only being able to cut materials straight down. All these components were assembled and bonded using acrylic Weld-On, ensuring a strong and clean finish.

The final pinball machine generally met our expectations in most key areas. The game successfully created a fun and engaging environment for players and onlookers. Gameplay was intuitive, and feedback like lights, sounds, and scoring made the experience rewarding and immersive. The machine attracted attention during testing and demonstration, proving that it encouraged group interaction and socialization. Several students noted that it bonded them with their classmates and would be a good break from academics. The game logic accurately handled mode transitions and the paddles were responsive and reliable. EE-specific references like the Patrick Mahomes ramp and the ‘EE’ targets resonated with the target audience, adding a sense of hilarity and familiarity.

While generally successful, some trade-offs and adjustments to the design were necessary. The flippers were intended to be solenoid activated, but this was not possible due to time and budget constraints. As a result, we transitioned to a purely mechanical alternative. The microcontroller and power board failed to properly function, resulting in broken IR beams which caused us to adjust game logic.

Before bringing it to market, we would fix these design challenges so as to enhance performance, durability, and overall user experience. This is discussed in more detail in Section 6.

2 Detailed System Requirements

To achieve a functioning pinball machine, both functional and aesthetic requirements had to be met. Functionally, there needed to be a mechanism that launched a pinball into the gameplay area that contained a multitude of stationary and moving objects. Then, the pinball needed to have two flippers at the bottom of the gameplay area that could be triggered by the user to push the pinball back into the gameplay area continuously toward the targets. In addition, the objects must have been able to be triggered for points to be added. The addition of these points created a score which was displayed to the user on a 7 segment. To track the start and end of the game, there needed to be a sensor that altered the system when the pinball first left the initial position and when it returned to mark the end of the game.

For launching the pinball into the gameplay area, a manual springer was needed. The springer was placed at the front of the box structure, with an opening that allowed the user to pull the handle while the spring mechanism pushed the pinball into the gameplay area. Near the springer, a set of IR beams were needed to detect if the pinball had broken the beam by moving past them, signaling that the game had started. For ease of coding, the game was split into three states: Idle Mode, Gameplay Mode, and Editing Mode. The IR beams triggered the transition from Idle Mode to Gameplay Mode and the transition from Gameplay Mode to either Idle Mode or Editing Mode based on the state of its beam.

Once in Gameplay Mode, the pinball was to move around the angled surface of the machine. To ensure the pinball stayed in the gameplay area, the two flippers at the bottom of the machine were controlled by the user to push the ball higher into the gameplay area. The user accumulated points during this time. Once the pinball fell past the flippers, the programming recognized the same IR beam pair used to start the game being broken again, signaling to the microcontroller that the game had ended. This part was essential for the programming to know what mode the game was in.

When transitioning to Idle Mode, the machine needed to go through a “game over” phase in which the LEDs flashed rapidly and the graphics were wiped clean to reveal a high score leaderboard. If the user’s score was high enough to be among the top three, the system entered Editing Mode. In this mode, they were able to enter their initials onto the screen using the two buttons on the front of the machine. The leftmost button cycled through the letters of the alphabet for a specific place in the initials, and the second button moved to the next place to the right until all three places were filled. Once exiting the high score screen, the score displayed was wiped to zero on the 7 segment display. Likewise, a rotating array of graphics displayed on the LCD monitor to confirm functionality and show an aesthetically pleasing, retro arcade inspired display.

These states and the tasks associated with them needed to be organized in the code using a FreeRTOS framework. During gameplay, the pinball encountered obstacles and targets. To gain points, the pinball had to hit one of the stationary targets or ascend up a ramp where another set of IR beams would detect its presence. The code kept track of these contacts and added points that simultaneously updated the 7 segment display with the current score. In addition to the targets and ramp, the gameplay area featured two motorized obstacles to make it substantially harder for the player to control the movement of the pinball and score points. One of these motorized obstacles was upright in the gameplay area with a 3D printed rod that will be attached on top. The motor alternated its spinning direction, causing the rod to unpredictably swing in circles

in the center of the gameplay area. This spinning obstacle near the bottom of the gameplay area was needed to add difficulty to hitting the ball upwards with the flippers. The second moving obstacle was a 3D printed hoop that moved on a jigsaw piece below it. There needed to be two metal contacts on the bottom of the 3D printed hoop such that the metal pinball completed the circuit and sent a signal to the microcontroller when it was hit. This gate obstacle moved back and forth horizontally to maximize the difficulty of aiming the ball into the hoop. This operated in front of the ramp to add difficulty to getting the ball inside the ramp, making the ramp obstacle the hardest in the machine. The ramp therefore awarded the most points.

Aesthetically, the machine required visual and audio outputs that enhanced the gameplay experience for the user. These requirements were a lot more flexible than the functional components, so our group chose aesthetic elements that mimicked the experience of a class pinball machine such as bright lighting, neon painting, and nostalgic audios when targets are hit. Also, EE-themed adornments were added to appeal to the target audience of electrical engineering students and professors. This included 3D printing the ramp so that it read a quote from a past EE professor: “If Patrick Mahomes can make a first down with a sprained ankle, you can make it up this ramp.” The two stationary targets were both decorated with an “E” so that next to each other the user reads “EE.” Both sides of the machine as well had clear lettering that says “EE” that allows you to see inside to the internal wiring. Although aesthetics came secondary to the functionality of the machine, it was still important to focus on making the game more tailored for the target audience so as to achieve our stated objective of bringing people together.

3 Detailed project description

3.1 System theory of operation

System Theory of Operation is described in terms of gameplay modes, mechanics, and flow.

When the system powers on and the ball is sitting on top of the plunger, the system is in Gameplay Mode. The user pulls back on the plunger, and the ball travels up the chute to the main portion of the playing field. This is when the user can use the left and right flippers on the side of the enclosure to keep the ball in play, attempting to hit the ball at the stationary and moving targets, as well as up the ramp. During gameplay mode, both motors are moving, meaning that the Spinner is interrupting ball movement and directing the ball up the playing field, and the Hoop is moving left to right across the playing field. The LCD has animations that are contributing to the spectacle of the

gameplay. The 7-Segment display updates as the stationary and moving targets are hit. If the user were to get the ball up inside and around the ramp, the ball would trip the IR beam there and trigger that input as well to create points. Whenever points are allocated to the user, a sound is played through the audio system, indicating to the user that they earned points without them needing to look up at the 7-Segment display.

When the ball falls in between the two flippers, it rolls to the right side of the playing field and triggers the second IR beam. This broken beam changes the state of the game. If the player has a new high score, the system enters Editing Mode. They can use two buttons on the front of the enclosure to enter their initials on the High Scores Leaderboard. The left button cycles through the alphabet, and the right button moves the cursor to the following letter.

Once the player enters all three letters in, the system enters Idle Mode. The system will also enter Idle Mode if the game ends and the user does not have a high score. In Idle Mode, the LCD displays the top scores, the 7-Segment displays the most recent score, and the motors move as they would during Gameplay Mode.

The following diagram describes the flow and interactions of the gameplay modes.

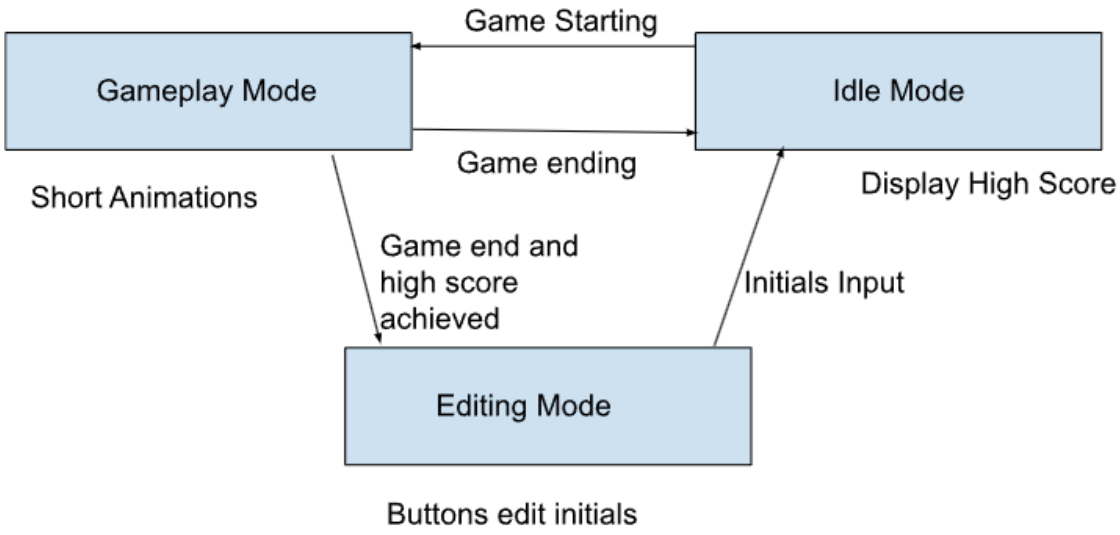


Figure 1. Gameplay Modes Diagram.

3.2 *System Block diagram*

The block diagram below shows our system's functions, relationships, and the interactions between the physical actions that trigger electric signals, and electric signal processing. The color key for descriptions of each interaction is at the top of the diagram.

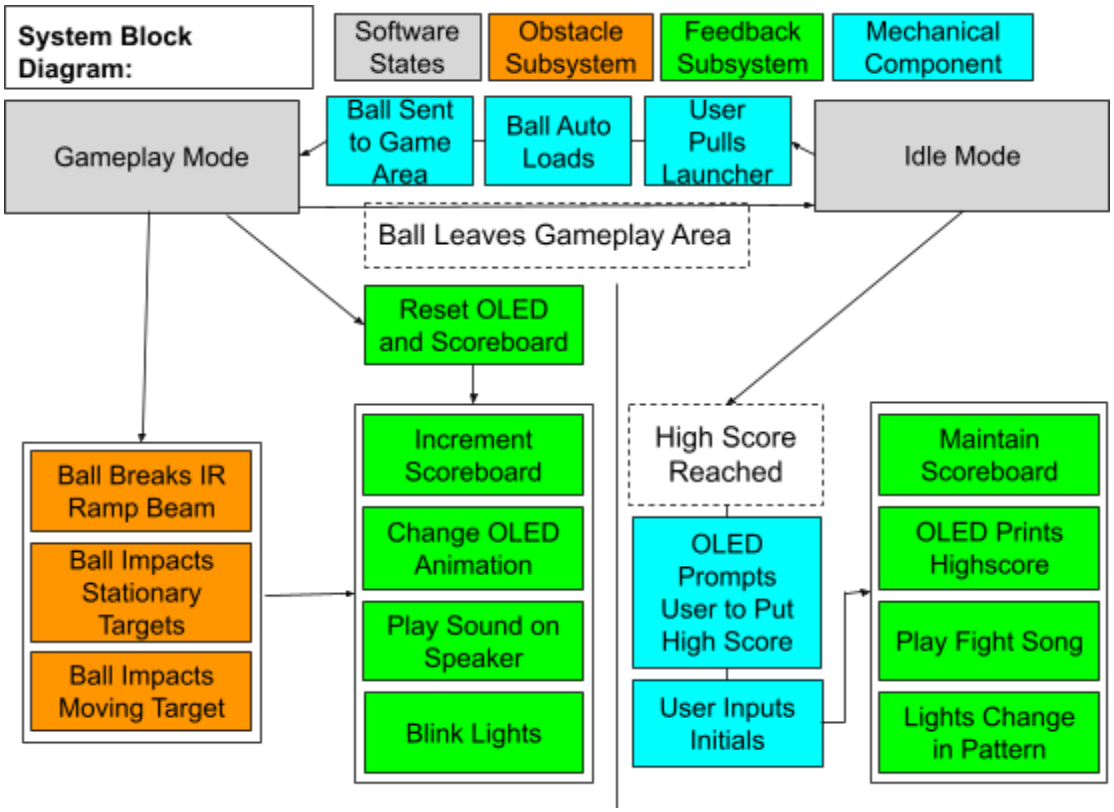


Figure 2. System Block Diagram.

3.3 *Feedback Subsystem*

The feedback subsystem contains audio, visual, and lighting elements to create a lively aesthetic to replicate the experience of vintage pinball machines. A speaker attached to a digital amplifier sounds different .wav files based on corresponding actions such as hitting a target and the game finishing. For visuals, the machine incorporates a 7 segment display that updates the score with each contact with a target and a Liquid Crystal Display (LCD) monitor to show changing geometric visuals throughout the gameplay. After the gameplay, if the user has one of the top three highest scores of all time, the LCD shows a screen where the user can input their initials to the system with their final score with the assistance of the electromechanical buttons to sift through the alphabet. Additionally, APA102C LED light strips adorn the outside rim of the gameplay

area to illuminate the area in varying ways depending on the timing of the gameplay. These elements of the feedback system allow for the game to be more interactive for the user and cause more excitement and nostalgia for all.

Speaker & MAX98357A Digital Amplifier

A I2S system integrates with the ESP32-S3 microcontroller to create audio feedback during the gameplay of the machine. The external aspects of this system are a MAX98357A digital amplifier and a Dayton Audio PC68-4 4-ohm resistance speaker. The writing diagram for the digital amplifier and speaker is shown below in Figure 3.

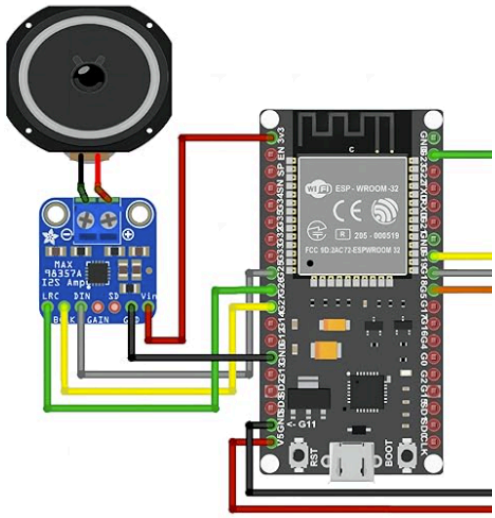


Figure 3. Wiring Diagram of the MAX98357A Digital Amplifier and Dayton Audio PC68-4 4-ohm Resistance Speaker.

The digital amplifier feeds 3.3 V from the power board. Besides the power and connections, the amplifier has a bit clock (BCLK), word select (WS), and data out (DOUT) connection to the microcontroller. BCLK acts as a clock signal that times each individual data bit sent or received to time the speed at which audio bits are clocked across the data line. WS determines which audio channel (left or right) the current data belongs to. DOUT outputs the actual digital audio data line to the amplifier one frame at a time.

In the FreeRTOS framework, there is an audio task that is responsible for playing the sounds. This task receives the name of the .wav file before calling the play_MONO_wav_file() function in our “Audio.h” file. This function opens the .wav file from the SPIFF storage that is stored in VS Code in the “data” folder. Then, the function

reads chunks of the audio data into a small buffer that is then sent over the I2S channel using the i2s_channel_write() function that is able to convert the contents of the buffer into audio output signals. This process is looped over and over until the entire contents of the .wav file are read and an audible output is generated. During gameplay when a target is hit, the corresponding .wav file is sent to the audio queue to play the appropriate sound. Our specific project uses a 32-bit data width that samples at a 48 kHz rate at 256 frames per buffer.

7 Segment Display

The Adafruit 1.2” 7-segment LED HT16K33 Backpack allows the user to see their updated score as the game is played. The device is pictured below in Figure 4.

Figure 4. Image of Adafruit 1.2” 7-segment LED HT16K33 Backpack.

This component requires a data (SDA) and clock (SCL) connection to the microcontroller on I/O pins and three connections to the power board: V_IO, +5 V, and ground. The SDA pin uses a serial data I2C interface connected to I/O Pin 48. The SCL pin connects to I/O Pin 47 as a I2C interface serial clock input. Both of these lines are connected to the positive supply via a pull-up resistor as required by I2C protocol to ensure proper logic levels. The start and stop conditions for the display are dependent on the transition on the SDA line while SCL is high. When the SDA line changes from high to low while the SCL is high, the display is in the start condition, while the SDA line changing from low to high while the SCL is high indicates a stop condition. This

mechanism can be better visualized in Figure 5. This behavior is important for framing correct data transmissions during initialization and gameplay.

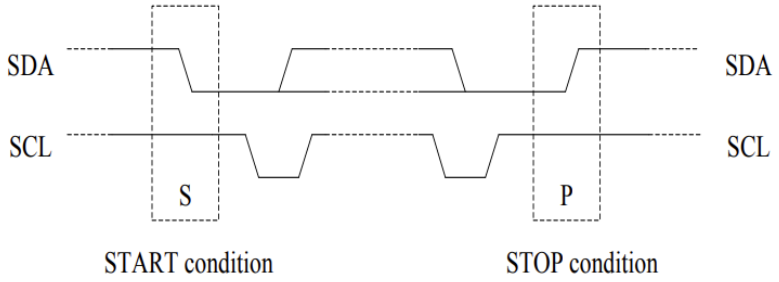


Figure 5. Start and Stop Conditions of the 7 Segment Display.

The Adafruit_LEDBackpack.h and Wire.h library is used to assist in the software to hardware interface for the 7 segment display. Since this select display uses 2 address select pins, the potential I2C addresses are 0x70, 0x71, 0x72, and 0x73. For our specific setup, we chose I2C address 0x70 when setting up communication with our HT16K33 driver. In the display task in the FreeRTOS code, the 7 segment is updated using the built-in functions matrix.println() and matrix.writeDisplay() to write the variable stored as “currentScore.” The function matrix.println() writes the “currentScore” value to the display buffer. The matrix.writeDisplay() transmits this value in the buffer to the physical display. A 250 millisecond delay is added to prevent this task from hogging the CPU and to prevent clogging the I2C bus. Also, 250 milliseconds is fast enough to allow for a nearly real-time score update to the human eye. Thus, these software decisions were made to ensure the best user experience while also preserving CPU efficiency and minimizing I2C bus load in the FreeRTOS multitasking environment.

Liquid Crystal Display (LCD) Monitor

The LCD used for this project is an Adafruit 3.5” 320x480 Color TFT Touchscreen Breakout that has individual RGB pixel control for 128x64 pixels. We use a SPI interface bus to communicate between the LCD monitor and the microcontroller. The hardware interface requires five main data signals: MOSI, CS, SCLK, DC, and RST. The MOSI pin transfers pixel and display data from the ESP32 to the display. The chip select (CS) channel enables or disables the display’s SPI communication. The serial clock (SCLK) provides the clock signal that synchronizes all of these data transmissions. The data/command select (DC) tells the display whether the current SPI transmission is a command or pixel data. The reset (RST) channel resets the display controller to the starting state.

Our code uses the Adafruit_GFX library and its accompanying functions in the display task. Table 1 below shows a list of the different functions used in changing the LCD display. These functions allow for the LCD display to be updated every 250 milliseconds with the latest information from the SPI bus.

Function	Purpose
tft.begin()	Initializes the SPI connection
tft.setRotation()	Rotates the display orientation
tft.fillScreen()	Fills the entire screen with a solid color
tft.setCursor()	Sets the starting point for the text drawing
tft.println()	Prints text at the current cursor location
tft.setTextSize()	Enlarges text proportionality
tft.setTextColor()	Sets the text drawing color

Table 1. LCD Display Functions Used from the Adafruit_GFX Library.

The LCD has two main functions for the pinball system. These both occur when the system enters Editing Mode after the completion of the game if the user achieves one of the top three scores of all time. When this occurs, the two buttons attached to the front of the machine are used for the user to control to input their initials. Button one cycles through the available characters to select the letter A-Z. Button two confirms the letter and moves to the next character position. This three character initial is then stored in three global character arrays in the RAM.

When the machine is in its Idle Mode while still powered, the drawHighScores() function is called in the display task that writes the three initials and high scores pairs below a red heading that reads “High Scores.” These initials and high scores are global variables that are updated in the gameTask() when a new high score is achieved. This function compares the current score of the user to the past three highest scores. For example, if the user achieves a score that would put them in the second highest of all time, the current second highest score would be copied to the place of the third highest score, and the second highest score would be rewritten with the new score. Then, the user would be prompted to put in their initials using the buttons to control the letters and positions of the initials. The LCD would react accordingly to the user inputting their

initials by drawing an underline at the current position being edited and constantly updating the letter as the user shifts through them.

During the Idle Mode, the LCD calls on predefined functions to draw lines, triangles, and squares in varying colors. This backdrop added to the vintage aesthetic of the machine while not being too distracting during gameplay for the user.

APA102C LED Light Strips

To bring an extra visual aspect of the machine to life, we implemented APA102C LED strips along the game play area. Two strips are serially connected to one another so that only one strip is needed to be connected to the microcontroller. The two strips together are comprised of 120 individual LEDs that are powered by 5 V from the power board. The two signal connections to the microcontroller are the serial clock (SCL) and data in (SDA) pins. These strips use SPI communication to control timing, the color palette, and animations.

LED functions in the coding are taken from the pre-defined FastLED.h library. The lights are first initialized based on the number of LEDs and a brightness value ranging from 0-255. The functions used from the FastLED.h library in the lighting code can be seen below with their corresponding purpose in Table 2.

Function	Purpose
FastLED.addLeds()	Initializes the LED strips
FastLED.setBrightness()	Sets the brightness of the LEDs
FillLedsFromPaletteColors()	Assigns a color to each LED
FastLED.delay()	Manages color themes and blending styles
switchLights()	Main function that updates animation each frame
ChangePalettePeriodically()	Cycles through visual effects every few seconds
SetupLights()	Initializes the LED hardware and sets the initial state

Table 2. APA102C LED Strip Functions from the FastLED library.

During the light task in the FreeRTOs framework, the LED strip updates the active color palette as time passes, shifts the color index to to animate the strip, updates the LED buffer with new colors, and sends the data to the LED. The LED buffer, called CRGB objects, represents the entire color state of all 120 LEDs in the strip. Each CRGB object contains three 8-bit values with each 8-bit segment representing the colors red, green, and blue. Each color is represented by a value ranging from 0-255 to allow for a wide range of colors to be shown in the strip. This buffer is 180 bytes total and is stored in the RAM.

There are four light states incorporated in the code, aptly named “LIGHT_IDLE,” “LIGHT_GREEN_BLINK,” “LIGHT_RED_BLINK,” and “LIGHT_CRAZY.” Changing the “currentLightMode” variable in the main code calls the light task in the “Lighting.h” function, which changes the lighting accordingly. The “LIGHT_IDLE” state causes a gradual rainbow cascading effect when the machine is powered on but not in Gameplay Mode. Then, when a point is scored via a stationary target, the “LIGHT_GREEN_BLINK” mode is called, causing a flash of green discernible to the human eye. Similarly, at the end of gameplay, the “LIGHT_RED_BLINK” mode is called to create this red flash to signal to the user that they lost. “LIGHT_CRAZY” is reserved for when the ball moves up the ramp, causing a crazy fast rainbow lighting effect to signify all the points gained. The light task can be seen in Figure 6.

```
void lightTask(void *pvParameters) {
  static LightEffect lastMode = LIGHT_IDLE;
  static bool isBlinkOn = true;
  static unsigned long lastBlinkToggle = 0;

  const unsigned long effectDuration = 1500;       // Total time to keep blinking
  const unsigned long blinkInterval = 250;         // Time between ON/OFF

  while (true) {
    unsigned long now = millis();

    if (currentLightMode != LIGHT_IDLE) {
      // New effect start
      if (currentLightMode != lastMode) {
        lightEffectStart = now;
        lastBlinkToggle = now;
        isBlinkOn = true;
        lastMode = currentLightMode;
      }

      // Handle blinking effects
      if (now - lastBlinkToggle >= blinkInterval) {
        lastBlinkToggle = now;
        isBlinkOn = !isBlinkOn;

        if (isBlinkOn) {
          if (currentLightMode == LIGHT_GREEN_BLINK) {
            fill_solid(leds, NUM_LEDS, CRGB::Green);
          } else if (currentLightMode == LIGHT_RED_BLINK) {
            fill_solid(leds, NUM_LEDS, CRGB::Red);
          }
```

```
      } else {
        fill_solid(leds, NUM_LEDS, CRGB::Black);
      }
      FastLED.show();
    }

    // End effect
    if (now - lightEffectStart > effectDuration) {
      currentLightMode = LIGHT_IDLE;
      lastMode = LIGHT_IDLE;
      fill_solid(leds, NUM_LEDS, CRGB::Black);
      FastLED.show();
    }

  } else {
    // Idle rainbow effect
    static uint8_t index = 0;
    fill_rainbow(leds, NUM_LEDS, index++);
    FastLED.show();
    vTaskDelay(pdMS_TO_TICKS(50));
  }

  vTaskDelay(pdMS_TO_TICKS(10));
 }
}
```

Figure 6. Light task in the “Lighting.h” header file.

3.4 Electromechanical Subsystem

To build an interactive gameplay arena that integrated smoothly with the software, we combined multiple electromechanical components into a cohesive subsystem. These components include a ramp with embedded IR beam break sensors, a spring-loaded launcher, microswitch stationary targets, two moving obstacles, and two buttons.

Ramp

A 5mm beam break sensor at the top of the ramp detects the ball passing through. The emitter contains two wires: one to 5 V and one to ground. The receiver has three wires: one to ground, one to 5 V, and one to an I/O pin on the microcontroller. The digital signal line goes LOW when the beam is received and HIGH when the beam is broken.

When triggered, it sends a 5 V digital signal to the microcontroller, which then gives scoring, audio, and visual updates. The code works such that it only sends one signal when the ball passes through by looking at a change in state. This ensures that each ball pass only triggers a single scoring event.

The ramp was custom designed in CAD with a small opening, creating a challenging gameplay experience. This is why it awards the user the highest amount of points and signals the specialized lighting effects. It features the quote “Patrick Mahomes ran a 1st down with a sprained ankle, you can make it up this ramp” as a callback to Professor Huang’s famed quote in Signals and Systems. Figure 7 below shows the CAD of the ramp.

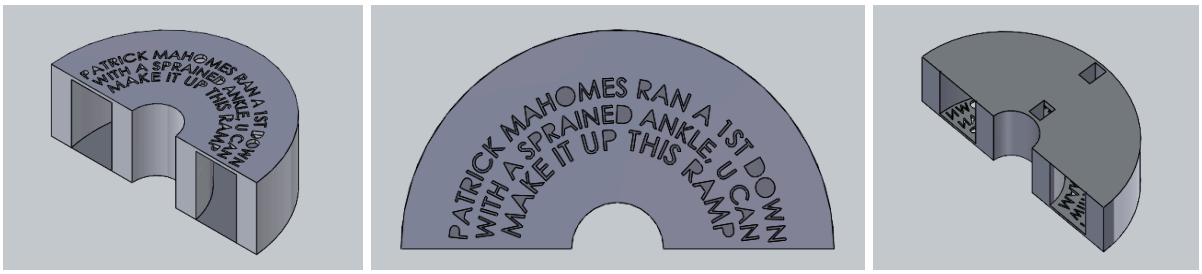


Figure 7. Design of Ramp with Embedded IR Beams

Interference from ambient light to the IR beam is not an issue as the IR beams are enclosed within the structure. Holes up the wall of the ramp allow for clean IR beam mounting and for concealed wire routing. The ramp is secured to the game play area with glue and the IR beam wires go through a hidden hole in the acrylic.

Spring Loaded Launcher

An IR beam break sensor by the launcher detects when the ball passes between the launcher and the gameplay area. The sensor is embedded into a cutout within a CAD-designed separator piece which divides the launcher runway from the gameplay area. The IR beams are glued into place, with wires running discretely down a hidden hole to the underneath of the gameplay area. Below are the CAD models of both sides of the assembly.

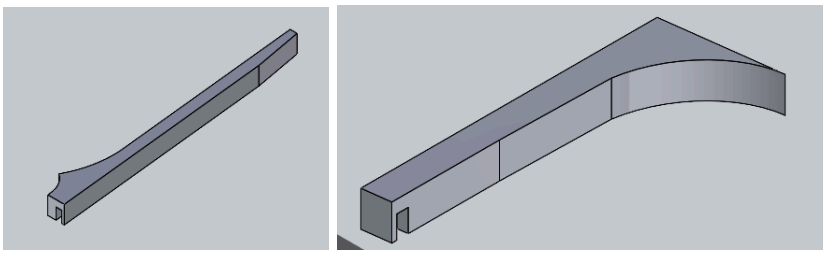


Figure 8. Design of the Chute Edges and IR Beam Holders

The state of the sensor distinguishes game states based on the pinball’s position. When the ball rests against the launcher, the system enters the Idle or Game State. When the

launcher is pulled back, the ball enters the loading area, breaking the beam. This signals the start of a new game. The code registers a game start only when the beam is restored, using state-change detection to avoid multiple triggers. Figure 1 on page eight shows a block diagram illustrating this logic.

The pinball machine incorporates the launcher from the 2022 design, and the casing was modeled to accommodate this. We chose this over an electrically actuated mechanism because it adds to the user experience and creates a vintage arcade experience.

Stationary Targets

We used two VM-5 Zippy Microswitches (5A/250Vac NO/NC), repurposed from the 2022 pinball machine. Each microswitch acts as a stationary target. They detect when the ball hits by registering a mechanical press that completes the circuit and sends a high signal. They use 3.3 V and require one signal line to the microcontroller each.

The target assembly included a frame attached to the microswitch, providing a surface for the ball to strike. They each have a cutout of an ‘E’ as a reference to our theme of electrical engineering. Figure 9 below is the CAD design of the target frames.

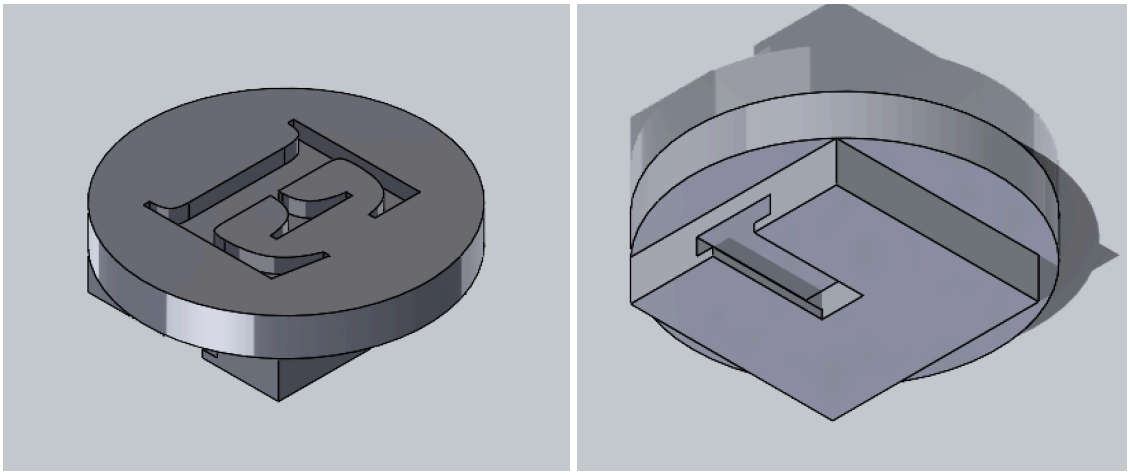


Figure 9. Design of the Target Frames

To prevent false triggers, a back boundary was designed in CAD to prevent the ball from hitting the back of the microswitch and sending signals to the microcontroller. These semicircular boundaries are mounted behind the targets. Figure 10 shows the CAD design of the back boundaries.

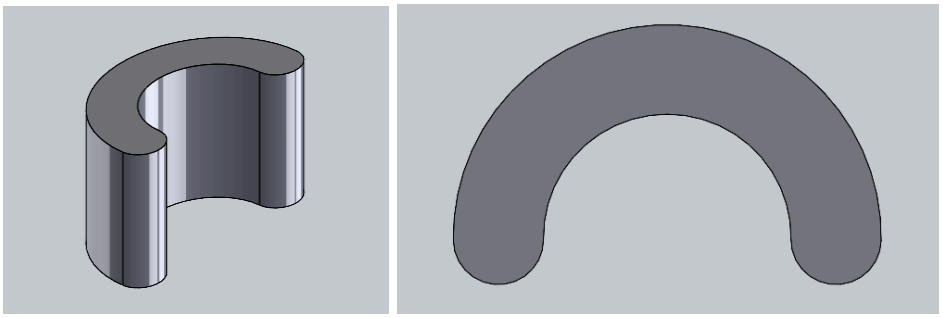


Figure 10. Design of the Target Back Boundaries

The micro switches are glued onto an angled 3D-printed L-shape piece on the underside of the acrylic game play area. The top part is glued onto the underside of the gameplay area. The microswitch is glued onto the longer flat surface. This way, nothing on the top of the gameplay board will get in the way of the user experience.

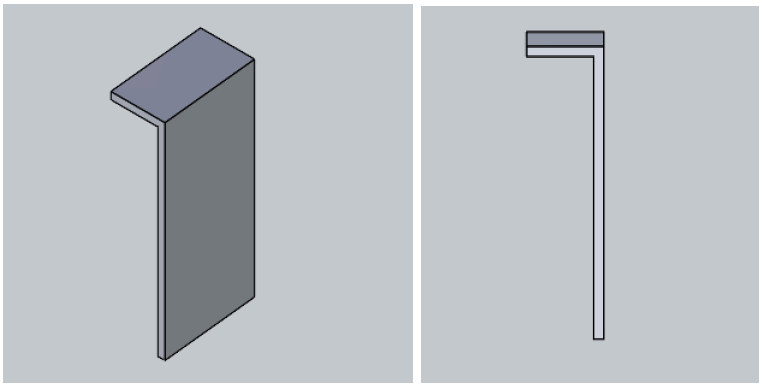


Figure 11. Design of the Switch Holder

We coded and tested the microswitches such that they detect when the target is hit. They register one hit only by looking at a change in state, as seen in the code in the appendix. Once a target is hit, it sends a signal to the microcontroller which then adjusts the player's score on the seven segment display. This also triggers the feedback system.

Moving Objects

There are two moving obstacles that are featured in our game design, referred to as the Hoop and the Spinner. The Hoop moves left to right during gameplay, offering a moving target for the player. The Spinner is a rounded beam towards the bottom of the gameplay area that rotates and operates as an additional source of movement for the pinball, as well as an obstacle for the user.

Both obstacles are powered by Mercury Motors Bipolar Stepper Motors (Serial Number: SM-42BYG011-25-090327); an image of the motor and its engineering drawing is shown below in Figure 12. These 12V, 330mA motors were found in a cabinet from an old senior design group. The motors are able to control both of the 3D printed obstacles.

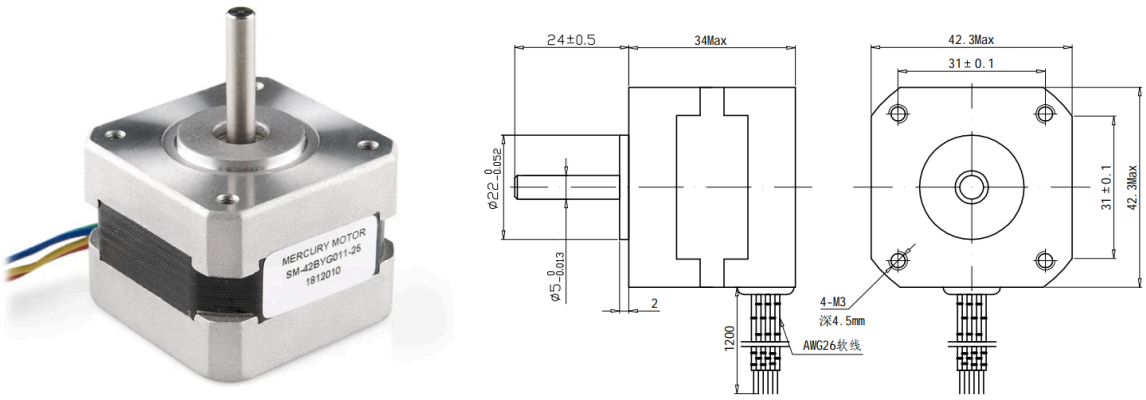


Figure 12. Stepper Motor Schematics

In order to simplify the signal processing, we decided to control the motors through driver boards, close to the motors' locations. We purchased Sparkfun Electronics EasyDriver (Serial Number: ROB-12779) because there was an accessible and easy to read wiring guide online, which can be found here. Our design used the EasyDriver's Step and Direction pins to control movement and direction of rotation. The MS1 and MS2 pins, often denoted as Logic1 and Logic2 in our code, control the size of the steps. The enable pin must be high to control the motor, minimizing power when the motor itself is not moving. Our design uses these 5 logic pins, a logic ground pin, and a 12V power supply. The motor connects through the original cables to 4 pins, corresponding to the two windings within the motor. Figure 13 is a schematic of the EasyDriver.

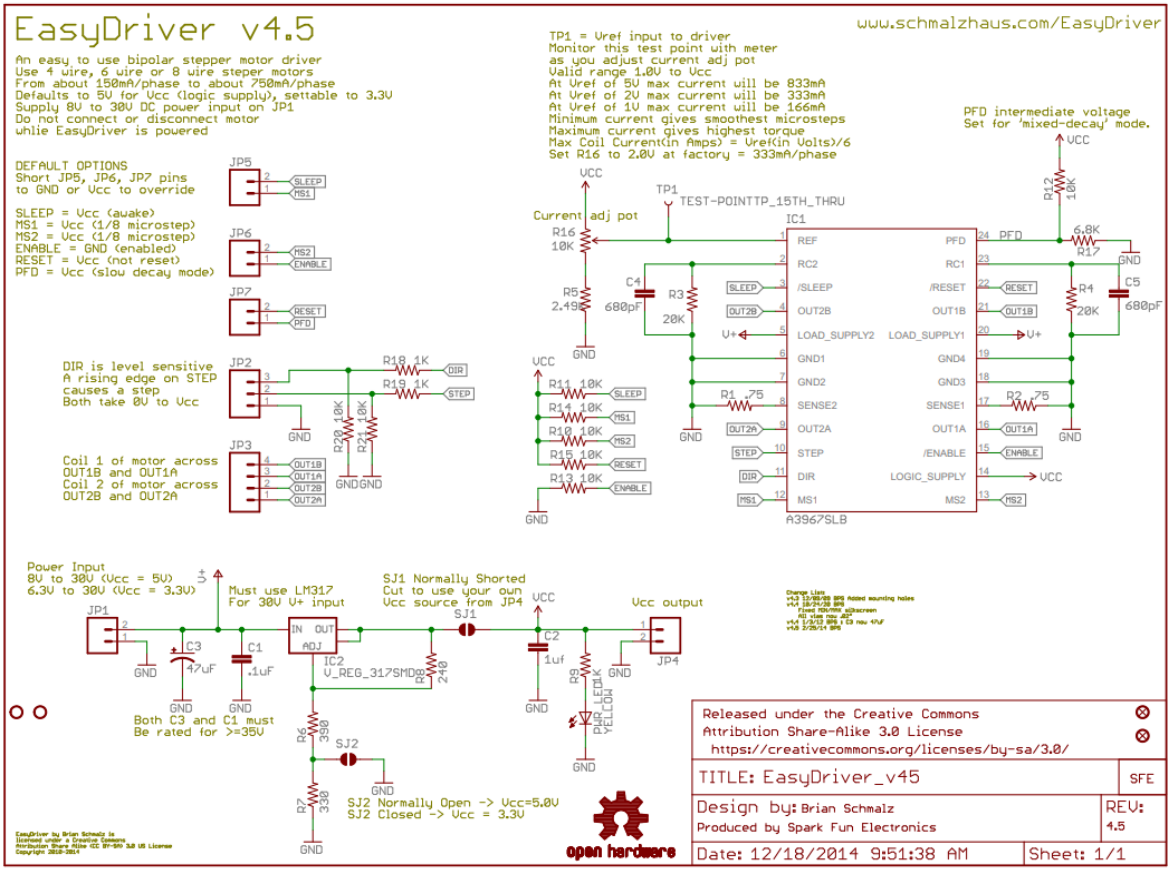


Figure 13. EasyDriver Schematics

The suggested library to work with the EasyDriver and the Stepper Motors is intended for Arduino boards, and did not translate well to the ESP32. The library used to control the motors is the ESP-Flexy-Stepper by pkerspe, which moves the motor using blocking calls.

The Spinner's 3D printed beam is attached directly to the spoke of the stepper motor. Screenshots of the CAD file are shown in Figure 14.

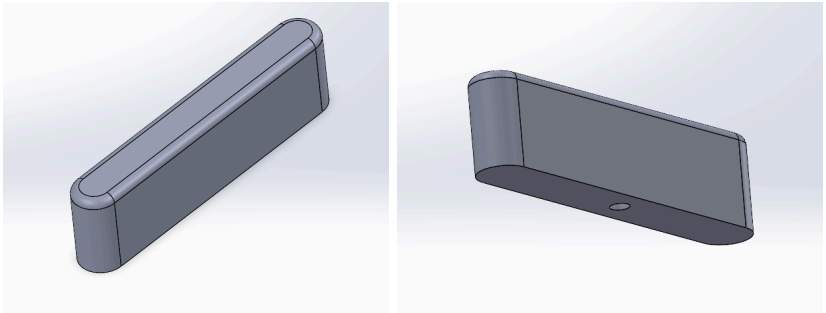


Figure 14. Design of the Spinner

In order to create horizontal motion from the rotational stepper motor, the Hoop uses a rack and pinion system to move the Hoop across the playing field. A rack is a beam with gear teeth attached to one side, which moves across one axis. The pinion is a gear that rotates around the source of movement for the system. The rotation of the pinion, when connected with the teeth of the rack, moves the rack back and forth across that plane. In our system, the pinion is attached to the stepper motor with the spoke of the stepper motor going through the middle. The pinion is shown below in Figure 15.

Figure 15. Design of the Pinion

The rack, a beam with teeth on the underside, sits on top of the pinion, with a hole for the pinball in the middle of the beam, as seen in Figure 16.

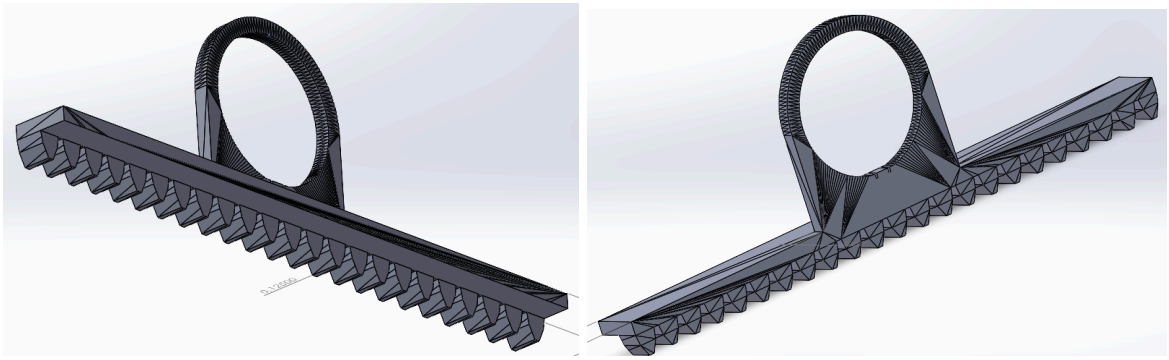


Figure 16. Design of the Rack

To ensure the rack does not fall through the slot of the playing field, a mount was designed for the beam to rest against, which is shown in Figure 17.

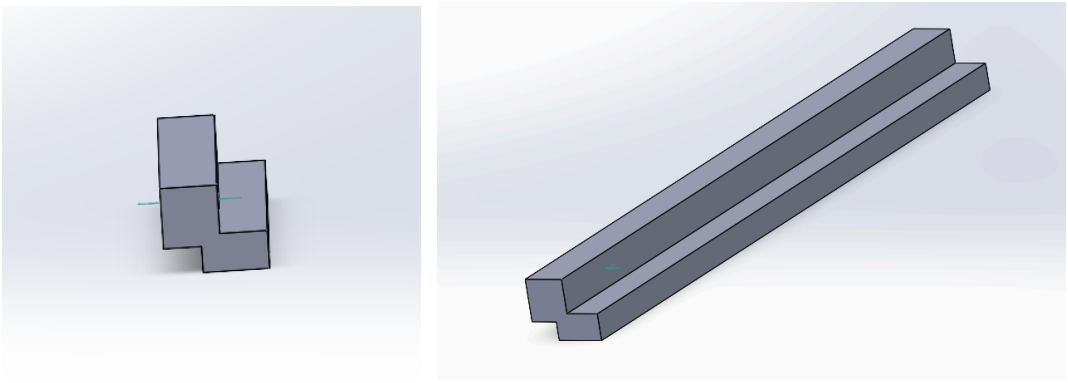


Figure 17. Design of the Rack Holder

To determine whether or not the ball rolls through the target, we attached wires to both ends of the divot at the bottom the circular cutout. When the pinball rolls through, it connects the circuit and sends an electrical signal to the microcontroller.

The last major difference between the Spinner and the Hoop occurred in its software. The Spinner is intended to rotate in one direction for the entirety of the gameplay. The Hoop must move back and forth, so the software included a boolean value that would switch from false to true or true to false after each movement. It would then move in the reverse direction the next time the task ran. The task is shown below in Figure 18.

```
void hoopMotorTask(void *parameter) {
  bool direction = true;

  while (true) {
    digitalWrite(enableHOOP, LOW);

    if (direction) {
      HOOP.moveRelativeInSteps(60);
    } else {
      HOOP.moveRelativeInSteps(-60);
    }

    digitalWrite(enableHOOP, HIGH);
    direction = !direction;

    vTaskDelay(200 / portTICK_PERIOD_MS);  // Wait 200ms between changes
  }
}
```

Figure 18. Hoop Software

The two motors were attached using custom CAD models shown in figure 19 below. They were glued in below the gameplay area on the flat top parts. The Spinner motor

holder allows the motor to sit upright, whereas the Hoop motor holder allows the motor to sit on its side.

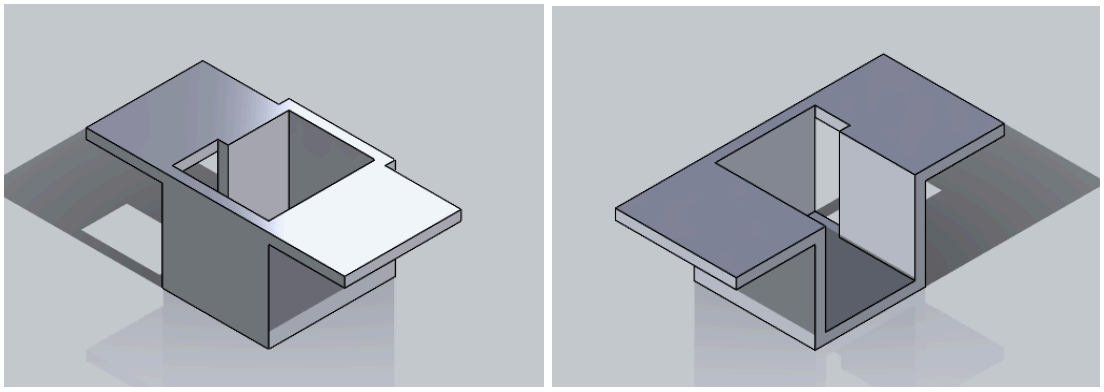


Figure 19. Design of Spinner (Left) and Hoop (Right) Motor Holders

Buttons

The two buttons at the front of the pinball machine structure are momentary push buttons that are essential for entering the initials of high score users while in editing mode. These buttons act as simple mechanical switches that determine the flow of current. When the button is not pressed, the button's internal contacts are open, meaning that no current can flow. Conversely, when the button is pressed, the internal contacts are closed, and current can flow. This creates digital logic based on whether the button is HIGH or LOW. When the button is pressed, the logic value is LOW, and vice versa. When the button is pressed, the microcontroller detects a change in the button state from the falling edge via the digitalRead() function.

Pull-up resistors are necessary here to prevent the input pin from floating when the button is not pressed, which would potentially increase the amount of noise in the system. This pull-up resistor ensures the HIGH state when the button is not pressed and a much cleaner transition to LOW when the button is pressed. These pull-up resistors are internal to the ESP32.

Each button has a certain task assigned to it in editing mode. Button 1 cycles through the alphabet for the necessary letters for the initials. Button 2 confirms when the user has hit the correct letter and then shifts to the next position in the initials. To get rid of any potential button bounce error, a 50 millisecond delay is added to the task.

Flippers Original Design

The original design of our flippers would follow traditional analog pinball machines, where two buttons on the side of the enclosure trigger a dual wound solenoid that either pushes the flipper quickly with high power draw or holds the solenoid up with low power draw. After researching alternative solenoids to purchase, it became apparent that a dual wound solenoid that would support a powerful solenoid was out of our budget. Therefore, the design would use the Adafruit solenoids in the EE Senior Design room (Product ID: 413). The drawing can be seen below in Figure 20.

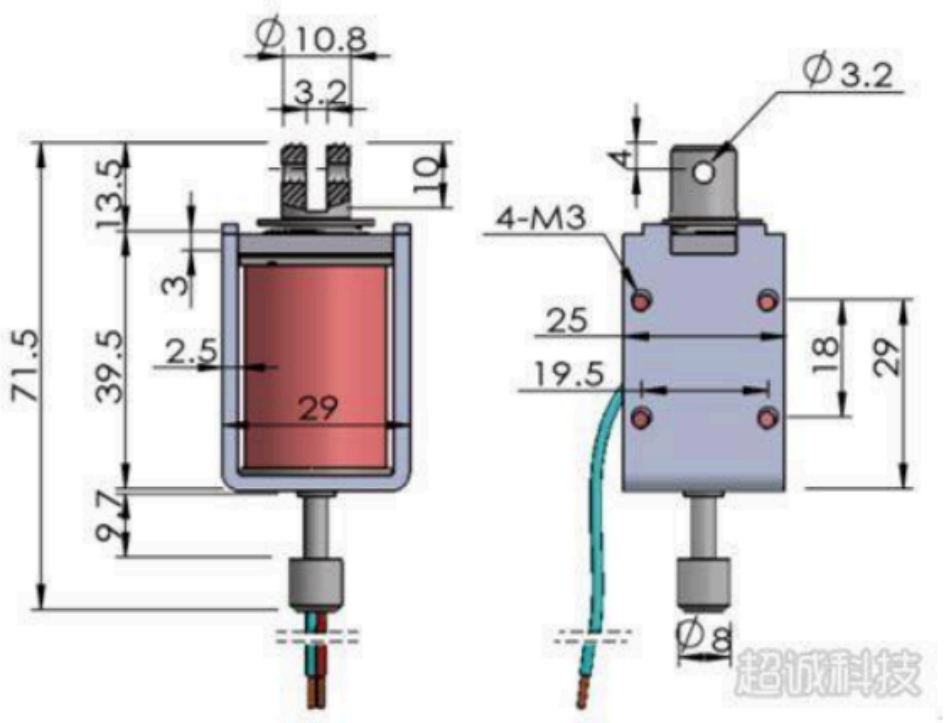


Figure 20. Solenoid Schematics

The initial design was that the solenoids would pull a 3D printed flipper horizontally, such that the pulling would push the flipper up. However, the solenoids do not pull horizontally with any force. Instead, the mechanism relies on the weight at the bottom of the beam that goes through the solenoid to induce motion. When that beam is horizontal, it does not move enough.

Therefore, the downward motion of the solenoid, in the negative z direction, would have to create some rotational movement in the x-y plane. To complete this rotational motion, the downward motion would pull against a rotary piece, connected via a ball joint rod

end that would allow the needed rotational motion, but would also push and pull the flipper. The intended assembly is shown below in Figure 21.

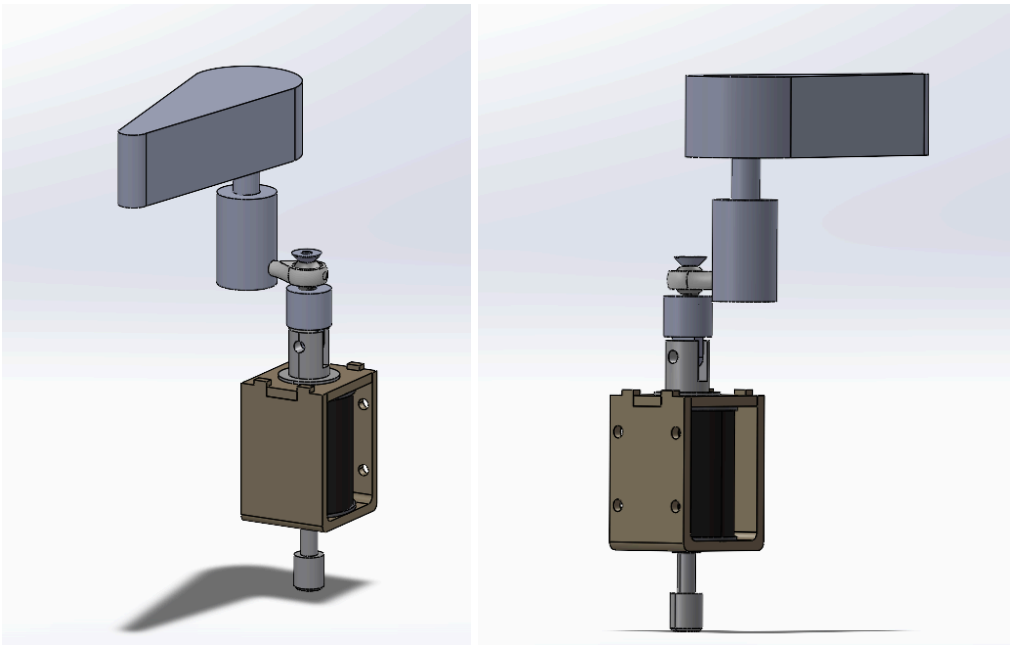


Figure 21. Design of the Solenoid Flippers

Instead of creating rotational motion, the activation of the system only pulled the flipper down into the playing field. Therefore, a mechanical design was used in the final iteration to demonstrate gameplay functionality of the whole system.

3.5 Mechanical Subsystem

Flippers Redesign

The mechanical redesign of the system was a lever and tension system, in which a beam protrudes from the gamefield enclosure, and the user then pushes the beam to push the flipper up. This was done using a cylindrical extrusion from the enclosure, a flattened beam flush with the gameplay surface, and a perpendicular lever connected to the flipper, as shown below in the assembly of the right flipper in Figure 22.

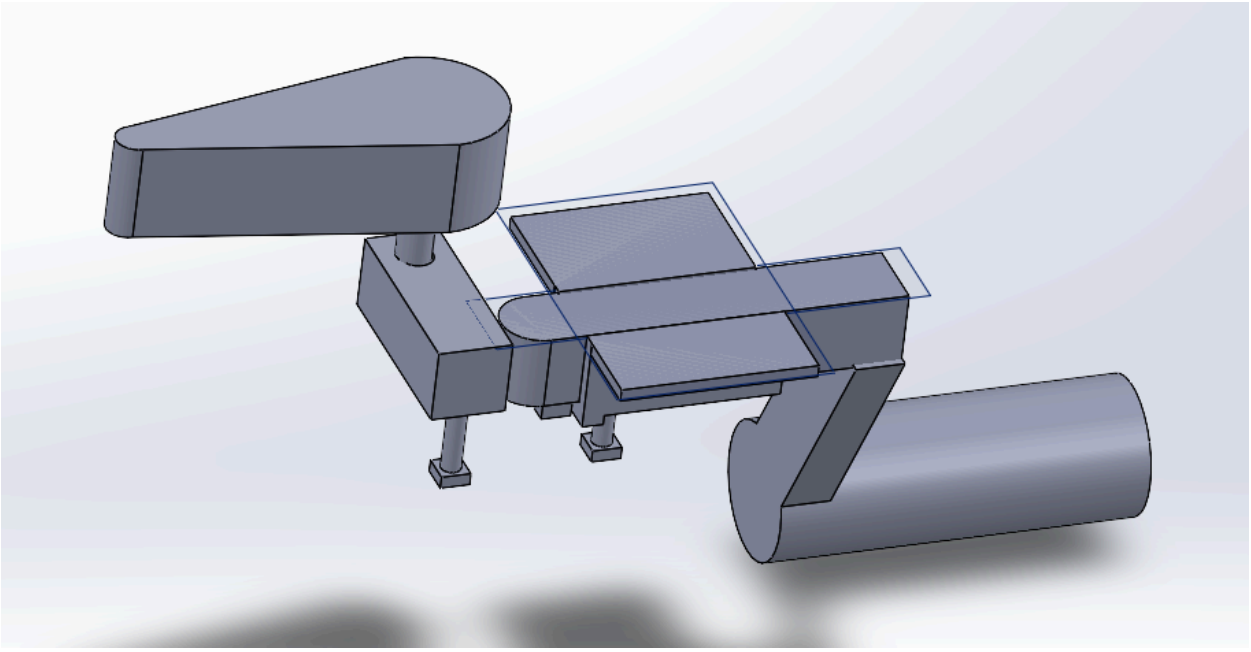


Figure 22. Design of the Mechanical Right Flipper

The extreme distance between the axis of rotation of the flipper and the protrusion from the enclosure was an advantage in designing the lever system. The left flipper had significantly less space to move, so the mount was moved to the inside of the flipper, as shown in Figure 23. To pull the flipper back down, like a traditional pinball slipper, a rubber band was attached to the rotational piece and a stationary piece, like the mount on the right flipper and the enclosure wall in the left flipper.

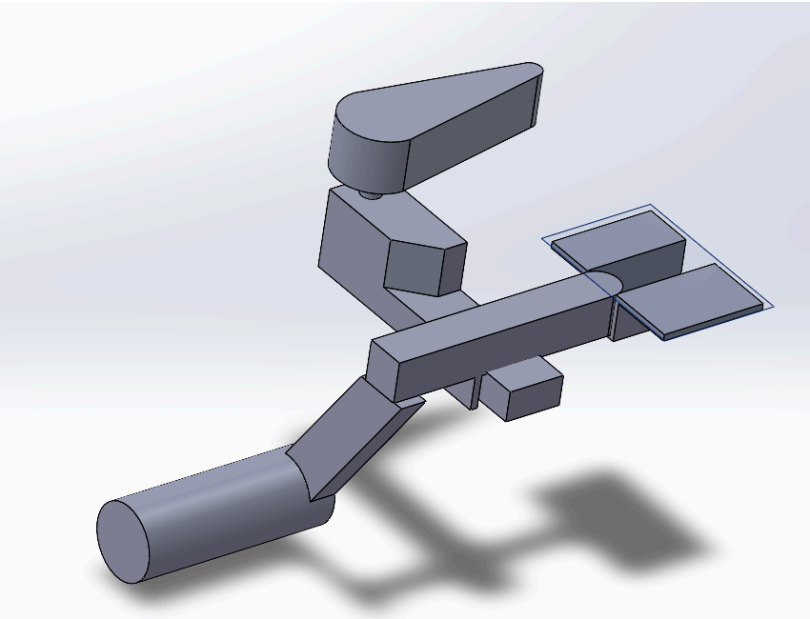


Figure 23. Design of the Mechanical Left Flipper

Enclosure

The physical enclosure was constructed using a 24” X 48” quarter-inch thick acrylic sheet, 3D-printed ABS plastic parts, and acrylic-specific Weld-On adhesive. The gameplay surface measures 12” x 22” and sits at a 6.5 degree angle to mimic traditional pinball dynamics. The overall machine dimensions are 12.5” x 25.5”. Acrylic panels were cut to size using a water-jet cutter at the Engineering Innovation Hub, while a laser cutter handled smaller, more precise cutouts for components and wiring channels. The DXF file for the laser cutter for the gameplay area is shown below in figure 24.

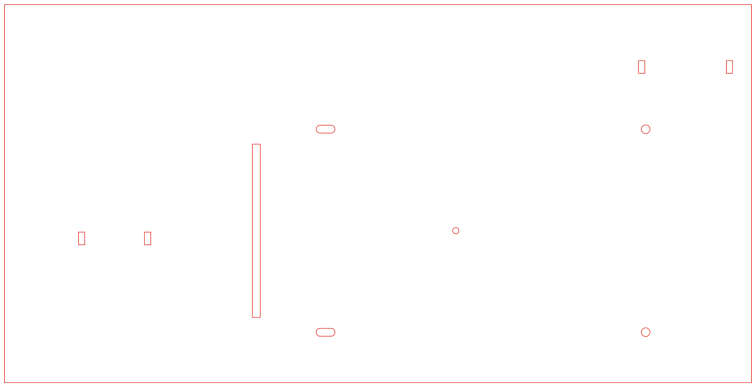


Figure 24. DXF File of Gameplay Area for Laser Cutter

Due to printer size limitations and the vertical cutting constraints of available tools in the Engineering Innovation Hub, the front casing was 3D-printed in two parts and later joined. Figure 25 below shows the full CAD model of this 3D-printed front part.

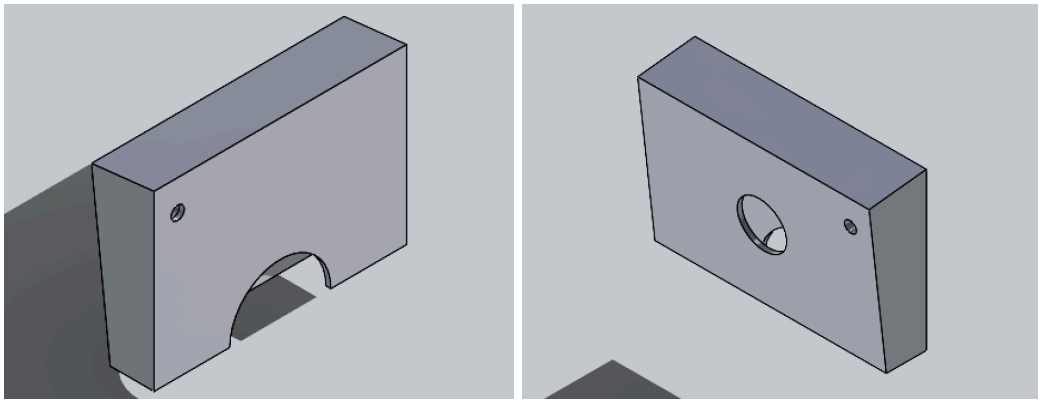


Figure 25. Design of Front Casing of Enclosure

All pieces were assembled with Weld-On to ensure a secure and seamless finish. After the acrylic and ABS plastic parts were joined together, the enclosure was spray painted

black. Tape was applied in the shape of “EE” on both sides before painting so as to add an aesthetic and electrical engineering element with users being able to see internal wiring. These letters were then outlined with bright acrylic paint for emphasis into our theme. The acrylic gameplay surface was painted with colorful stripes while the taped over striped allowed for spaces where the LED lights could shine through. The components on top were also painted with bright acrylic paint. Everything was sprayed with Modge Podge for durability and finished look.

The corner rounders and stationary boundary near the plunger are designed to enhance gameplay by improving ball movement. The stationary boundary by the plunger helps guide the ball during launches, keeping that area separate and ensuring that the ball consistently follows a controlled path up the playfield. Both components are 3D-printed and glued into the gameplay surface.

Figure 26 below is a picture of the finalized pinball machine, showing the final enclosure.

Figure 26. Finalized Pinball Machine

3.6 Power Subsystem

The power subsystem's main component consists of the separate power board, which is directly powered from a 120 VAC wall adapter. This converts to a 12 VDC connection to power the rest of the board and the system as a whole with a max of 5 amps. The extent of the power is modeled below in Figure 27. Power flows from the 12 V connection directly to the motors used to power both the spinner and the moving gate. This has a max of 2 amps with both connections needing about half an amp, falling far below this threshold. The 12 V connection is also inputted into the 5V voltage regulator. This voltage regulator allows a maximum of 3 amps. However, this total is not reached. The 5 V directly connects to the 7 segment display and the LEDs. These connections do not draw much current with the 7 segment display needing 80 mA and the LEDs needing 60 mA. The MCU board is powered by this 5 V connection. Once on the MCU board, a separate voltage regulator regulates down to 3.3 V at 1 amp maximum. This is used for nearly everything on the board except more LED connections. Power flows directly to the digital amplifier to power the sound and the LCD to power the graphics.

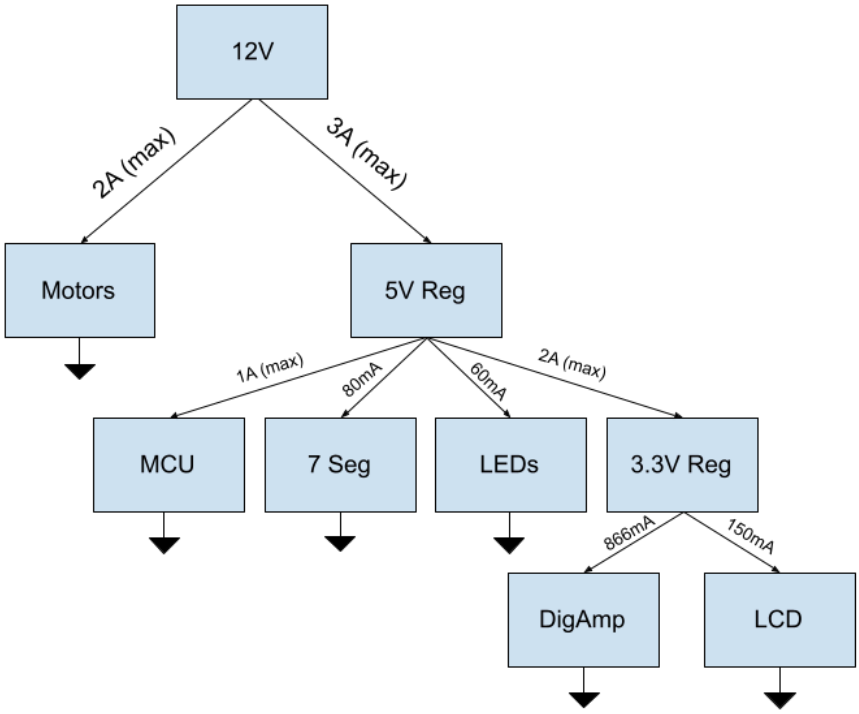


Figure 27. Power Flow Diagram

3.7 Interfaces

The most important part of our interface design is that significant portions of the signal processing are localized to the subsystems themselves. One instance of this is seen in our power system; there are three levels of power needed for proper operation and implementation of this design. Therefore, the power distribution system was localized to the power supply, a power distribution board, and power and ground JSTs coming outside of the power board. Instead of having one board and 4 levels of power, our localized power system simplified board design and minimized the number of wires going to and from our microcontroller board.

Another instance of this localized signal processing is the motor drivers. The EasyDriver boards, which drove the stepper motors, had their own H-bridge that funneled high voltages and high currents to the stepper motors but were controlled by digital inputs and outputs by the microcontroller. Driving loads running on 12 V and 0.66 A on the same board as systems that ran on 3.3 V and 80 mA could be dangerous for the loads. If our traces were misaligned, the loads and the driver systems could have been damaged. This is another instance of localized signal processing that protected the viability of all subsystems in the machine.

The other significant localized signal processing was the required digital amplifier for the sound in our feedback system. The MAX98357A Digital Amplifier board takes in I2S signals from the microcontroller, converts the signals, and then amplifies them before sending those signals to the 4 ohm speaker. Though moving the signal processing away from the microcontroller meant that the timing was difficult to achieve, the short distance between the speaker and our board minimized signal degeneration between the two.

The three subsystems and components discussed above used additional signal processing boards that were separately connected to loads and the microcontroller. Below in Table 3, the interfaces between components, subsystems, and components are detailed. This table emphasizes the connections between components, the power board, the microcontroller, and any additional boards.

Component	Power Board, (Voltage)	Microcontroller	Number of Connections	Additional Hardware
7-Segment	Yes, (5V)	Yes	4	
Audio	Yes (3.3V)	Yes	3	Speaker, Digital Amplifier
Buttons	Yes, (3.3V)	Yes	2	
LCD	No	Yes	5	
LED Lightstrips	Yes, (5V)	Yes	2	
IR Beam	Yes, (5V)	Yes	2	
Microswitches	Yes, (5V)	Yes	2	
Moving Target Trip	Yes (3.3V)	Yes	1	
Stepper Motors	Yes, (12V)	Yes	12	EasyDriver Board
Solenoids	Yes, (12V)	No		

Table 3. Wired Interfaces Separated by Component, Power Connection, Microcontroller Connection, and Required Additional Hardware

The original solenoid design for the flippers has been included in the list of components and interfaces, despite the change to a mechanical system in the final design. The mechanical design of the flippers had no wired interfaces with the other components or subsystems within the design.

4 System Integration Testing

4.1 Describe how the integrated set of subsystems was tested.

Due to the number of subsystems and different parts in the total design, an iterative approach was used for integration. First, each component of the machine was tested individually to ensure the component was working and to write basic code that will be useful later. Different group members were in charge of individual parts. Once all the parts were shown to work individually, it was important to then upgrade the complexity

of the coding incrementally until the part meets our final expectations of it independently. Then, it was time to add elements together to get them to work in tandem. To simplify this process, the targets, IR beams, and buttons were simulated with buttons on a breadboard at first. In total, six buttons were used to represent the following components: the two stationary targets, the IR beams at the start of the game, the IR beams in the ramp, and the two buttons to input initials. This process allowed us to simulate the states of the game – Idle, Gameplay, and Editing – by manipulating the IR beam buttons. Then, hitting the target and IR beam buttons representing the ramps would simulate earning points in the system. This system allowed for us to test the peripherals, such as audio, lighting, the 7 segment display, and the LCD monitor, without having to first set up the entire gameplay arena.

Using this button system, first the lighting and audio were synced. This allowed for the gameplay experience to be enhanced by causing the gameplay area to light up green and a dinging sound to occur when a point is scored. Then, when the game ends, the arena will flash red while a traditional “game over” sound plays. Once this was achieved, the 7 segment display was added to this setup so that it would increment when a stationary target is hit or the ball moves up the ramp. Then, the 7 segment display would be cleared once the gameplay is over in preparation for the next game. Next, the LCD was integrated, which was the hardest of the peripherals due to the wire length needed to connect the display to the microcontroller interfering with the SPI communication. This required altering the frequency of communication to the LCD to below 8 MHz. This allowed the screen to work with the code at the expense of some time lag. Once the LCD was able to work, we tested the logic on the buttons to allow for the user to input their initials and embellished the design on the LCD with different colors and underlines.

Now that these peripherals were synced with the gameplay logic, the next step was to integrate the actual stationary targets, buttons, and IR beams into the system. This was done outside of the gameplay area for ease of movement. This part of the project was fairly simple since the code was previously working on the buttons. Thus, any problems in the troubleshooting process would likely be due to hardware issues. This allowed us to discern that both sets of the IR beams were broken and needed to be replaced. However, besides this, the integration of the stationary targets, IR beams, and buttons was relatively seamless.

This completed the first part of the iterative process. This process was not yet complete because the coding was not yet converted to the FreeRTOS framework, leading to lags, and these components not yet being integrated into the pinball box. Next, the code was converted into the FreeRTOS format where elements such as lighting and the gameplay

logic acted as tasks to increase efficiency. With the new FreeRTOS code, all the components worked in tandem outside of the pinball box.

To tackle transferring the components of the design to the pinball structure, the subsystems were wired one by one and then tested with the previous code just for that device to ensure proper wiring. This proved more and more difficult as more components were added, making the wiring more complicated and causing connections to easily detach. Thus, adding a new subsystem often meant cycling back to test previous ones as well to ensure the setup was still working properly. The code tested in these instances were simple Arduino code and not yet in the FreeRTOS framework. Once this was achieved, next it was time to slowly implement the RTOs tasks into the code. This proved straightforward until the final task was to be added- audio. When the audio task was attempted to be implemented, a watchdog timer was set off. Nothing could be done in the timeframe of this project that was able to stop the timer from being set off, even attempting to disable it. Thus, the audio has to be abandoned in the final iteration of the project.

The last step was to integrate the flippers and spring launcher into the final design so that the gameplay experience of the user could be tested. The spring launcher worked as it should and was able to launch the pinball with enough force to move it up the causeway and down into the gameplay area. However, some issues arose when implementing the flippers into the design. One flipper often got caught when used by the user, causing the flipper to have special handling when used to prevent it from breaking. The final fix required moving the handle forward before pushing it to the side to work. The second flipper was more powerful and easier to use, making it better for gameplay in general.

4.2 Show how the testing demonstrates that the overall system meets the design requirements

This testing was vital to ensure that all the initial design requirements we set at the beginning of the project were met. The most important requirement was that the pinball machine created an enjoyable gameplay experience for the user. Although not all the components that were intended to be implemented were in the final iteration, our team was able to create a functional gaming experience. Although the flippers were accident-prone, the user was able to launch the pinball into the arena using the launcher and maintain it there using the flippers. The stationary targets were successfully linked with the 7 segment so that hitting them incremented the user’s score. Unfortunately, the IR beams in the ramp were fried after they were already glued into the ramp attached to the machine, making them difficult to swap out. Thus, the ramp was not able to be used to increment the score at all. However, the IR beams at

the initial position of the ball were able to be swapped out when they stopped working, so we were able to electrically toggle between the different game states still. This allowed for ease of control of the lights and LCD between the different states. The spinning and hoop motors were working but were not shown on demonstration day as the constant noise and occasional hoop motor sometimes getting stuck could be a nuisance for the user. Additionally, the users during testing were receiving much lower scores than anticipated, so having the moving obstacles would have likely been too difficult. As noted in the section above, the audio was not able to work due to a watchdog timer issue in the coding. The audio was an aesthetic part of the design, thus not being as important as components that affected the overall game functionality. Overall, the user was able to play a functional game of pinball with some aesthetic elements such as the lights and LCD working properly. Despite not all the subsystems being able to make it to the final product, the users were able to play the game with ease and our machine was recognizably pinball.

5 Users Manual/Installation manual

5.1 How to install your product

Our product becomes pre-installed. All the user has to do is plug in the power board to the wall.

5.2 How to setup your product

The product automatically turns on once it is powered. For simplicity, there is no on/off switch. To set up the machine for gameplay, all the user has to do is place the metal pinball in front of the plunger.

To start the game play, simply pull back the launcher and release to send the ball into the game area. This action automatically starts the game, activating the scoring system, sound effects, and lighting. For best results, ensure that the ball is fully situated on the launcher before pulling back.

5.3 How the user can tell if the product is working

The product has different features that indicate that it is working based on the state the machine is in. When first plugged in, the machine will become powered, initiating the LED strip effects and the geometric shapes on the LCD display. Once the pinball is shot into the gameplay area via the plunger, the machine will enter Gameplay Mode. In a properly functioning Gameplay Mode, hitting the stationary objects will increment the score on the 7 segment display. If the pinball falls below the flippers, the game will end.

If the score achieved by the user is in the top three of all time, then the machine will now be working in Editing Mode. Editing Mode is working if the LCD monitor then flashes with a screen that allows the user to input their initials to be stored in the RAM with their high score. The two buttons at the front of the machine box will be used to input the correct initials. The pink button when pressed will sift through the alphabet to the correct initial while the blue button will confirm when the correct letter is reached and move to the next position in the initials.

5.4 How the user can troubleshoot the product

If the product is still under warranty, the customer can return the product to the pinball machine team to be repaired. The warranty on this product expires May 2nd, 2025 at 5:00PM EDT.

If the warranty has expired, the user can try finding the source of the problem. Due to the multiple electrical connections, the most likely issue is an improper connection. First, the customer should identify the component that is causing the issue. For example, if the liquid crystal display (LCD) went black and was becoming unresponsive to the gameplay, then it would be best to trace the electrical connections from the LCD to the microcontroller and power to see if they are still intact. To do this, it is best to unplug the machine and then very carefully flip the machine on its side with caution to ensure no other connections are harmed. Then, follow the wires protruding from the back of the LCD to both the microcontroller and power to ensure they are still connected. If the wires are disconnected, reconnect them following the provided microcontroller pinout. If the wires are still properly connected, use a multimeter to ensure that the component is drawing enough current, and that it is receiving voltage.

6 To-Market Design Changes

Due to the time and budget constraints given with the Senior Design class, the best version of the end product could not be fully realized. Before bringing it to market, we would make a few key changes that would enhance performance, durability, and user experience. These include:

1. **Upgrade to 48V electrical flippers**
   Commercial pinball machines use higher voltage flippers for easier user interaction and more responsive gameplay. Our pinball machine currently uses mechanical flippers that operate on a tension based system. One notable change that could be made is to control the flippers fully electronically. Currently, the flippers mechanical system works, but it could be much smoother without the

tension based system. This could not be done as the cost of power solenoids were too expensive. Plus, the flipper system would work much better at a higher voltage, typically this voltage would be around 48V. This also would be too expensive to effectively utilize with our budget as other parts would have been scrapped.

2. **Add more dynamic LCD displays**
   The current LCD setup offers limited visuals as our group was more focused on getting it to a functional state. A commercial version would feature more displays that can offer a more immersive and exciting experience.

3. **Stronger theme**
   Aesthetically, the pinball machine would most likely need to stick closer to a specific theme as most pinball machines that go to market are heavily themed to increase their appeal. In the commercial version, our machine would have a stronger connection to the Notre Dame Electrical Engineering Curriculum. We would feature one of our professors as a final boss on the LCD and have the audio tie into funny moments specific to the program.

4. **Modular gameplay**
   For a more varied experience that would keep players coming back, we would add modular gameplay components that could click in and out. For instance, the spinning motor two pronged attachment could be swapped out for different sizes and shapes.

5. **Add a bottom panel for structure and wire containment**
   The current prototype lacks a bottom surface. This does not pose an issue when the machine is being used on a solid surface, but it makes the machine difficult to transfer while increasing the risk of shifting and damage. For a commercial version, we would add a sturdy, hinged bottom that can be easily taken on and off for maintenance if necessary.
6. **Board Changes**
   The board layout could be rearranged to better suit the actual needs. The current labeling of the pins is wrong. For example, the SCL and SDA pins are not labeled as IO47 and IO48. The board does work just not with the current assignments of pins. To ensure that the manufacturing can be done properly, these all would need to be relabeled. Rearranging the board in a more aesthetically pleasing way with more spacing would be helpful too.

7 Conclusions

In the end, the pinball machine was completed, and it does play pinball! The machine was able to be integrated together effectively and some lessons were learned.

The mechanical system effectively pushes the pinball as needed, and that was a very important part of the design to figure out early on. The box sizing, the angle of the play area, the plunger size, the flipper power, the ball sizing, and any other mechanical components were important to determine before figuring out anything else as everything depended on these mechanical parts.

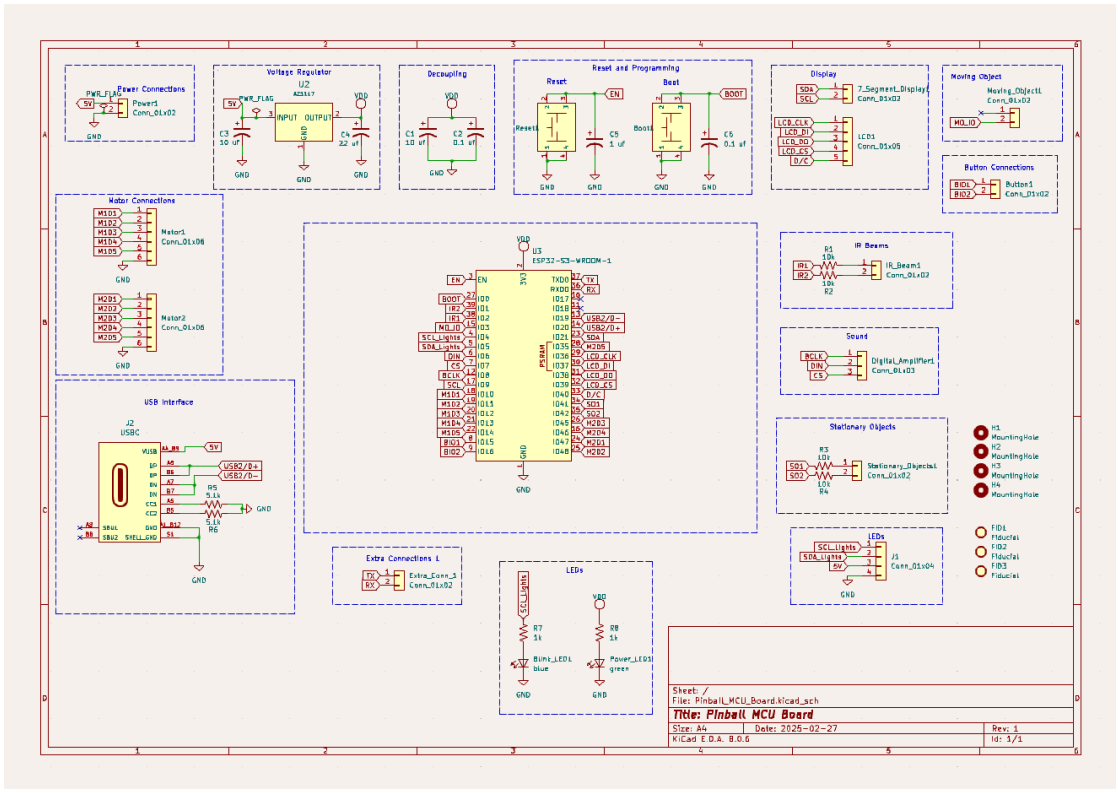
The aesthetics of the pinball machine are truly amazing and are an important part of what makes the gameplay experience fun and memorable. Ease of use and ease of replayability are important as well. Without the graphics, especially the high score display after the game is over, this would be a pretty sorry excuse for a pinball machine. This especially helped mask issues of reliability in terms of point scoring or other systems that were quite finicky.

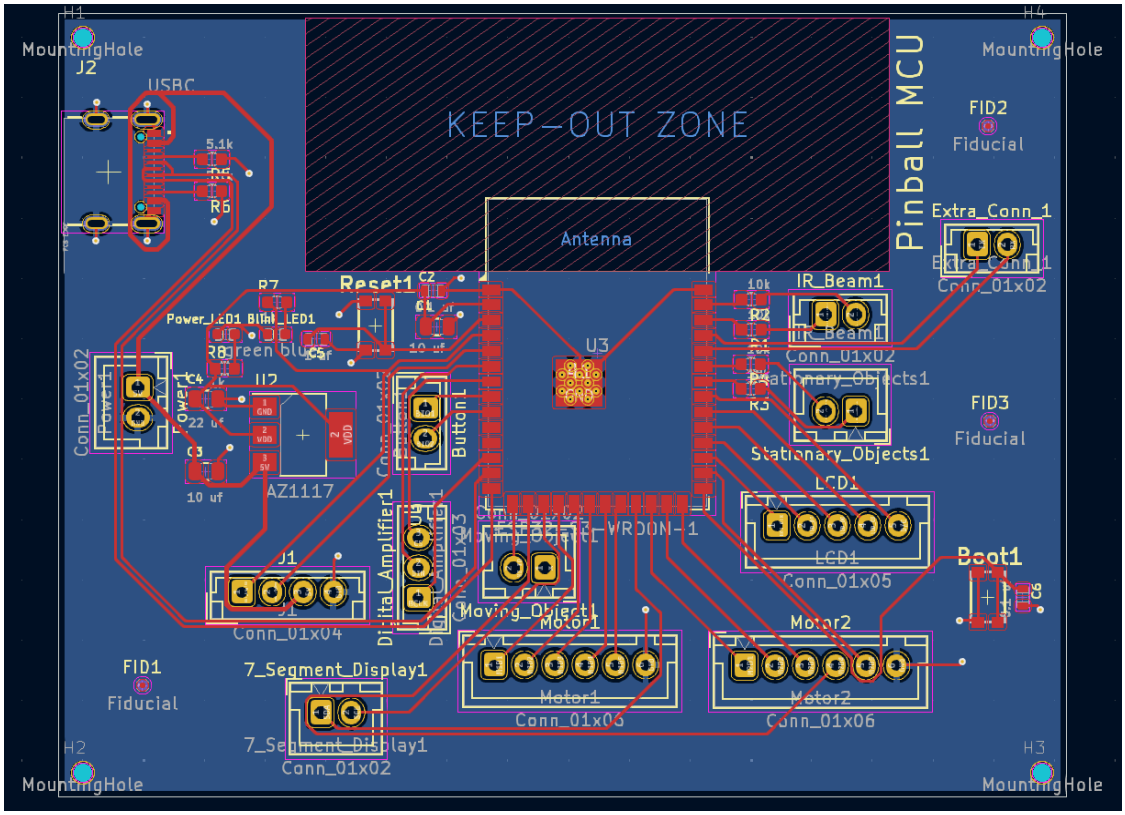
Integrating the system as a whole was much harder than anticipated as the expectation was to be able to combine all of the code together, but it proved far more complicated than that. If we were to do this all over again, we would make sure to not split up into individual subsystems per person as heavily and make sure each person is working across different subsystems to ensure better integration when the time comes. Segmenting the workload just creates bigger problems come integration time. Also, finding a way to better run wiring than JSTs would lead to a lot less reliability issues from connectors becoming dislodged.

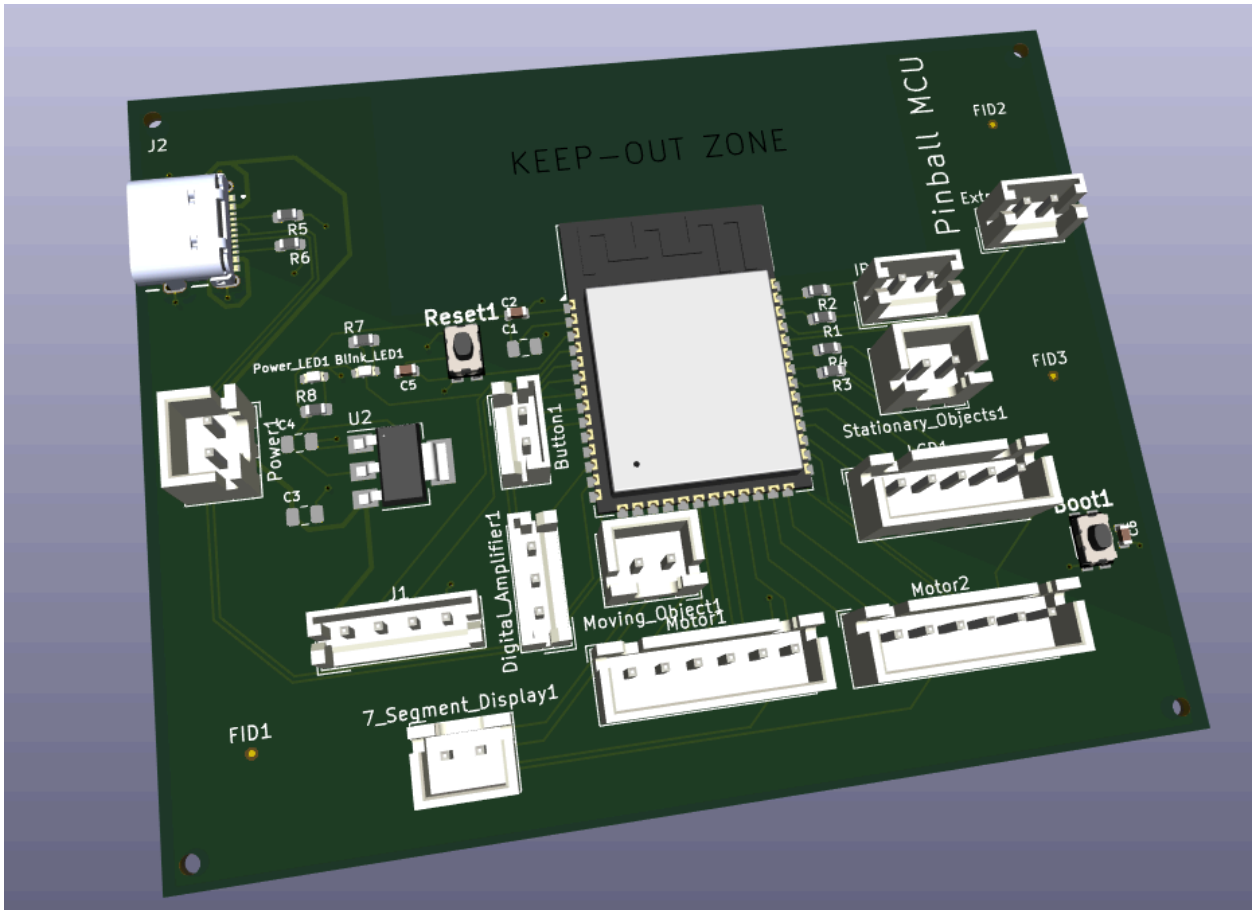
8 Appendices

Appendix A: Board Schematics and Layouts and Completed Machine

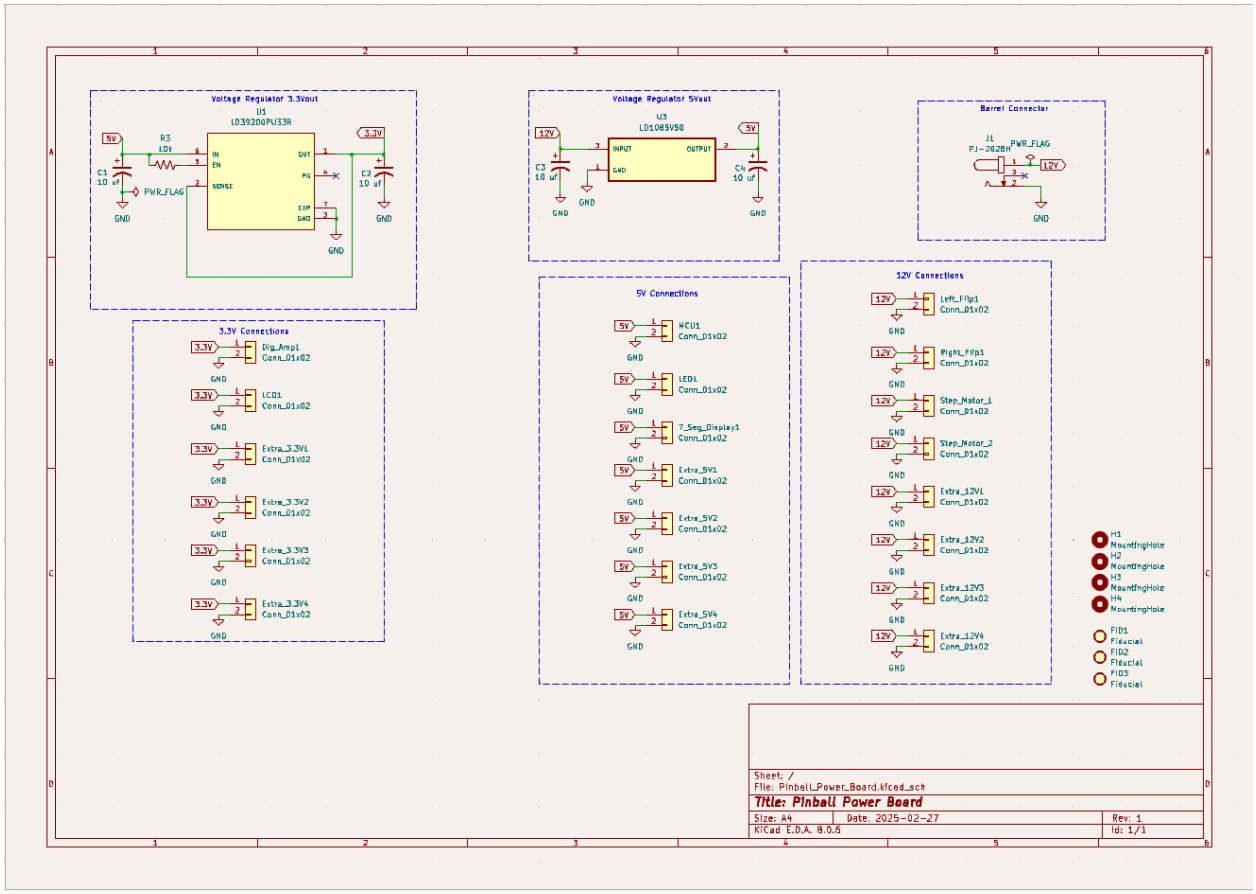
MCU:

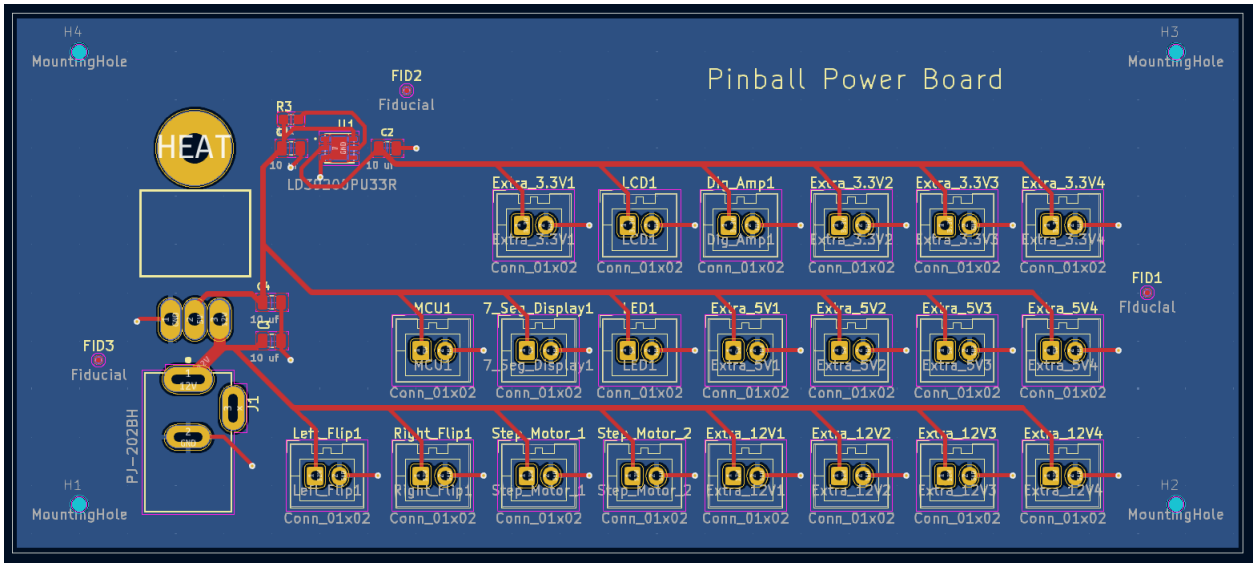

Power Connections
Voltage Regulator
Decoupling
Reset and Programming
Display
Moving Object
Button Connections
Motor Connections
IR Beams
Sound
USB Interface
Stationary Objects
Extra Connections 1
LEDs
Sheet: /
File: Pinball_MCU_Board.kicad_sch
Title: Pinball MCU Board
Size: A4
Date: 2025-02-27
KiCad E.D.A. 8.0.6
Rev: 1
Id: 1/1

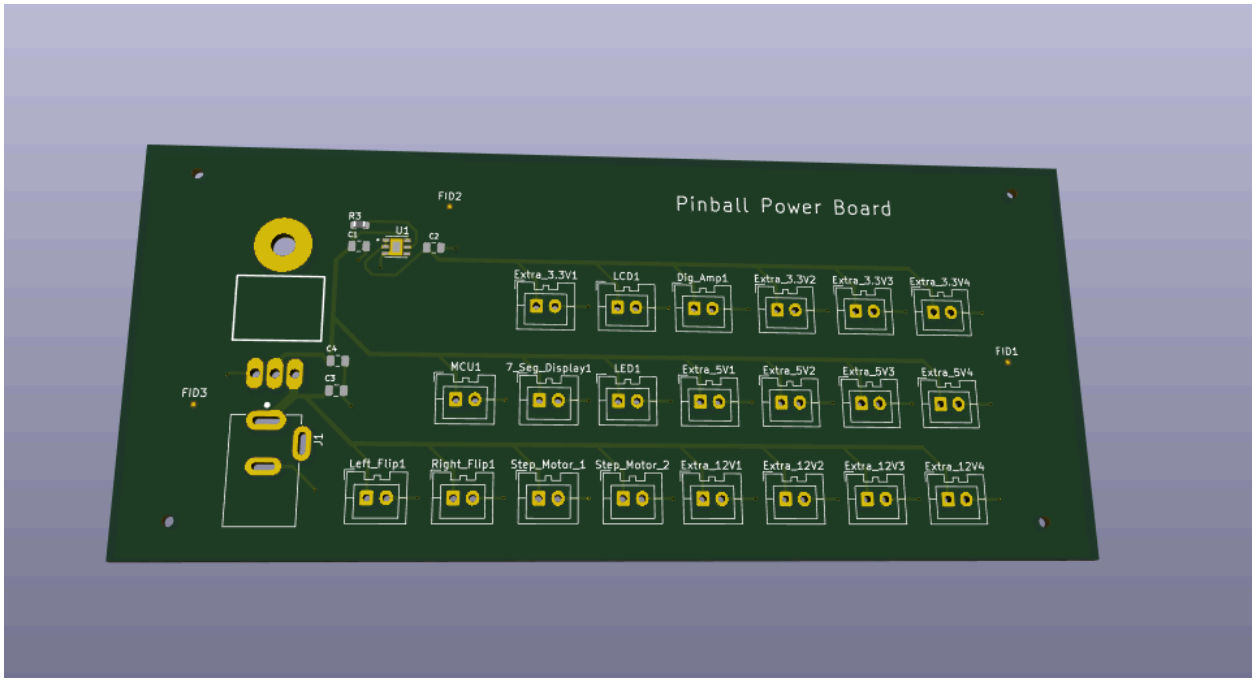

KEEP-OUT ZONE
Antenna
Pinball MCU
MountingHole
FID1
Fiducial
FID2
Fiducial
FID3
Fiducial
Extra_Conn_1
IR_Beam1
Stationary_Objects1
LCD1
Boot1
Motor1
Motor2
7_Segment_Display1
Moving_Object1
Button1
Power1
U2
AZ1117
U3
ESP32-S3-WROOM-1
J2
USBC


KEEP-OUT ZONE
Pinball MCU
J2
R5
R6
R7
Reset1
C2
C1
Power_LED1
Blink_LED1
C5
R8
C4
U2
C3
Power1
Button1
R2
R1
R4
R3
FID2
FID3
Stationary_Objects1
Boot1
J1
DigitalAmplifier1
Moving_Object1
Motor1
Motor2
7_Segment_Display1
FID1

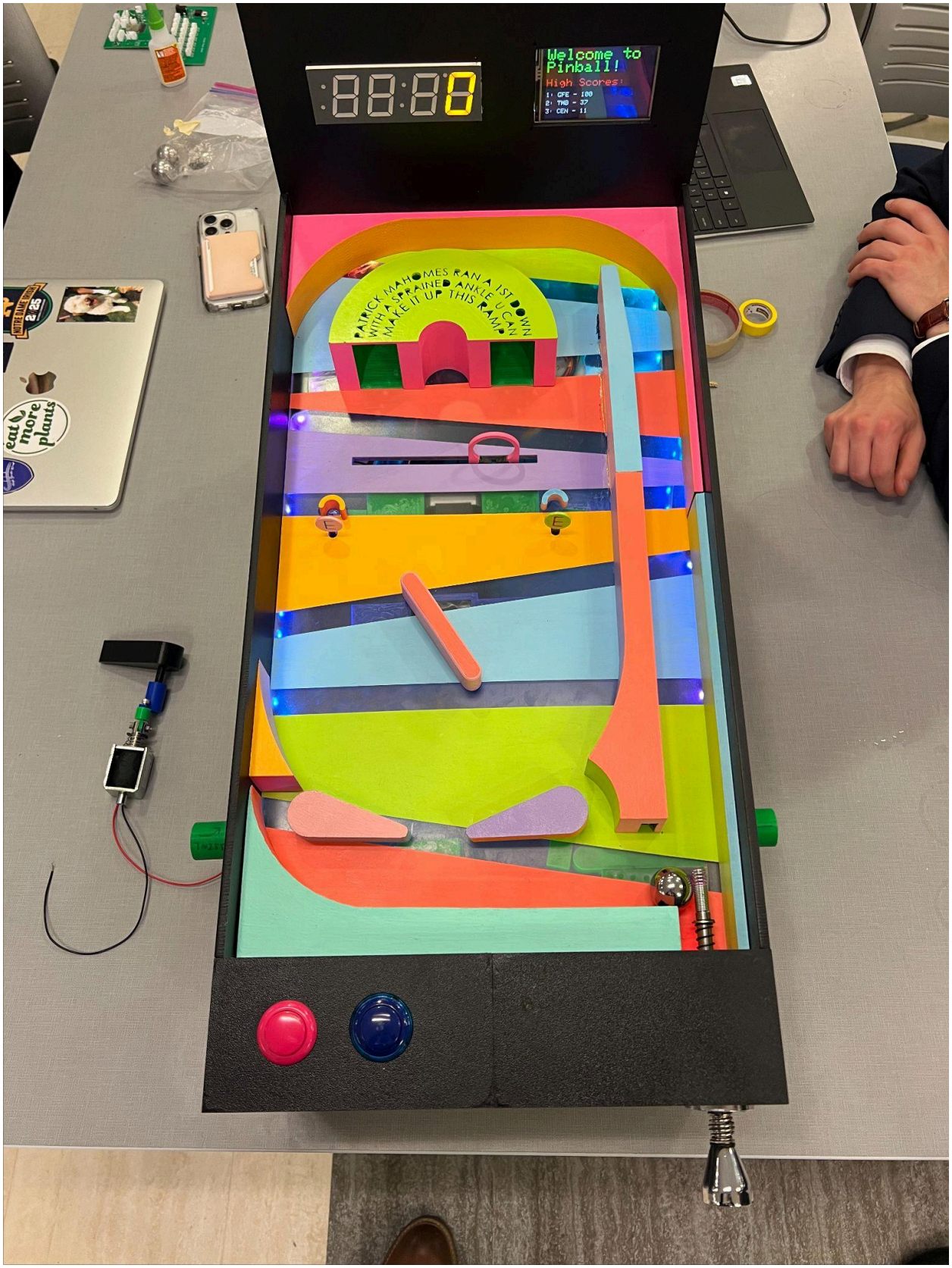
Power Board:


Voltage Regulator 3.3Vout
Voltage Regulator 5Vout
Barrel Connector
3.3V Connections
5V Connections
12V Connections
Sheet: /
File: Pinball_Power_Board.kicad_sch
Title: Pinball Power Board
Size: A4
Date: 2025-02-27
KiCad E.D.A. 8.0.6
Rev: 1
Id: 1/1


Pinball Power Board
MountingHole
Fiducial
HEAT
PJ-202BH
Extra_3.3V1
LCD1
Dig_Amp1
Extra_3.3V2
Extra_3.3V3
Extra_3.3V4
MCU1
7_Seg_Display1
LED1
Extra_5V1
Extra_5V2
Extra_5V3
Extra_5V4
Left_Flip1
Right_Flip1
Step_Motor_1
Step_Motor_2
Extra_12V1
Extra_12V2
Extra_12V3
Extra_12V4
Conn_01x02


Pinball Power Board
FID2
FID1
FID3
R3
C1
U1
C2
C4
C3
J1
Extra_3.3V1
LCD1
Dig_Amp1
Extra_3.3V2
Extra_3.3V3
Extra_3.3V4
MCU1
7_Seg_Display1
LED1
Extra_5V1
Extra_5V2
Extra_5V3
Extra_5V4
Left_Flip1
Right_Flip1
Step_Motor_1
Step_Motor_2
Extra_12V1
Extra_12V2
Extra_12V3
Extra_12V4

Completed Pinball Machine:




PERIPHERALS
Welcome to Pinball!
High Scores:
Hello
Alison
Huang

Appendix B: Complete Software listings

main.cpp:

```
#include "Arduino.h"

// includes for LCD
 #include <SPI.h>
 #include "Adafruit_GFX.h"
 #include "Adafruit_HX8357.h"
 #include "LCD_Graphics.h"

 // includes for audio
 #include <FS.h>
#include <SPIFFS.h>

#include "Audio.h"
#include "SPIFFS_tasks.h"

#define FORMAT_SPIFFS_IF_FAILED true

// includes for motors
#include "Motors.h"

// includes for lighting
#include "Lighting.h"

  // includes for 7 segment
 #include <Wire.h> // Enable this line if using Arduino Uno, Mega, etc.
 #include "Adafruit_LEDBackpack.h"
 #include "Adafruit_GFX.h"

 Adafruit_7segment matrix = Adafruit_7segment();

 // Use hardware SPI (on Uno, #13, #12, #11) and the above for CS/DC
 Adafruit_HX8357 tft = Adafruit_HX8357(TFT_CS, TFT_DC, TFT_RST);

 // SoftSPI - note that on some processors this might be *faster* than
hardware SPI!
```

```
 //Adafruit_HX8357 tft = Adafruit_HX8357(TFT_CS, TFT_DC, MOSI, SCK,
TFT_RST, MISO);

 char initials1[4] = {'T','M','B','\0'};
 char initials2[4] = {'C','E','N','\0'};
 char initials3[4] = {'M','R','N','\0'};
 char tempInitials[4] = {'A','A','A','\0'};
 char letters[28] =
{'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','
S','T','U','V','W','X','Y','Z','-','\0'};

 enum State {IDLE, EDITING, GAMEPLAY};
 State currentState = IDLE;

 int letterIndex = 0;  // Current letter being edited
 int charPos = 0;      // Current position in initials (0-2)

 bool lastButton1State = HIGH;
 bool lastButton2State = HIGH;
 bool button1State = HIGH;
 bool button2State = HIGH;

 bool lastIRState = HIGH;

 volatile bool targetHit = false;

unsigned long lastDebounceTime1 = 0;
unsigned long lastDebounceTime2 = 0;
const unsigned long debounceDelay = 50;

// cursors to write highscores
int score1_cursorY;
int score2_cursorY;
int score3_cursorY;
int letterUnderlineX;
int letterUnderlineY;

int motorCounter = 0;
// motors
ESP_FlexyStepper spinner;
```

```
ESP_FlexyStepper hoop;

// high scores (change to be empty values)
int score1 = 37;
int score2 = 11;
int score3 = 8;
int currentScore = 0;   // most recent score
int scoreToEdit;

// housekeeping
int graphicsCounter = 0;
bool buttonsPressedAndReleased = false; // Add this as a global variable

void highScore(){
  unsigned long start = micros();
  tft.setCursor(0, 0);
  tft.setTextSize(7);
  tft.setTextColor(HX8357_GREEN);
  tft.println("Welcome to Pinball!");
  tft.setTextSize(3);
  tft.println();
  tft.setTextColor(HX8357_RED);
  tft.setTextSize(5);
  tft.println("High Scores:");
  tft.setTextColor(HX8357_WHITE);
  tft.setTextSize(3);
  tft.println();

  score1_cursorY = tft.getCursorY();
  // tft.fillRect(0, score1_cursorY, 200, 30, HX8357_BLACK); // Adjust
width/height as needed
  tft.setCursor(0, score1_cursorY); // Move cursor back
  tft.print("1: ");
  tft.print(initials1);
  tft.print(" - ");
  tft.println(score1);

  tft.setTextSize(2);
  tft.println();
  tft.setTextSize(3);
```

```
  score2_cursorY = tft.getCursorY();
  // tft.fillRect(0, score2_cursorY, 200, 30, HX8357_BLACK); // Adjust
width/height as needed
  tft.setCursor(0, score2_cursorY); // Move cursor back
  tft.print("2: ");
  tft.print(initials2);
  tft.print(" - ");
  tft.println(score2);

  tft.setTextSize(2);
  tft.println();
  tft.setTextSize(3);
  score3_cursorY = tft.getCursorY();
  // tft.fillRect(0, score3_cursorY, 200, 30, HX8357_BLACK); // Adjust
width/height as needed
  tft.setCursor(0, score3_cursorY); // Move cursor back
  tft.print("3: ");
  tft.print(initials3);
  tft.print(" - ");
  tft.println(score3);
}

void updateEEBallBounceAnimation() {
  static int x = 30;
  static int y = 30;
  static int dx = 3;
  static int dy = 2;
  static int radius = 20;
  static uint16_t color = HX8357_YELLOW;

  // Clear the previous ball
  tft.fillCircle(x, y, radius + 1, HX8357_BLACK);

  // Update position
  x += dx;
  y += dy;

  // Bounce off edges
  if (x - radius <= 0 || x + radius >= tft.width()) {
    dx = -dx;
```

```
    color = random(0xFFFF);  // change color on bounce
  }
  if (y - radius <= 0 || y + radius >= tft.height()) {
    dy = -dy;
    color = random(0xFFFF);  // change color on bounce
  }

  // Draw the new ball
  tft.fillCircle(x, y, radius, color);
  tft.setCursor(x - 12, y - 6);
  tft.setTextSize(2);
  tft.setTextColor(HX8357_BLACK);
  tft.print("EE");
}

// Button 1 - Cycle through letters
void handleButton1Press(int initial) {
  if(currentState == GAMEPLAY){
    Serial.println("🔴 Ball In Holding Area (Beam Broken)");
    currentState = IDLE;
    Serial.println("Entering IDLE Mode");
    tft.fillScreen(HX8357_BLACK);
    digitalWrite(LED_BUILTIN, LOW);
  }
  if (currentState == EDITING) {
    letterIndex = (letterIndex + 1) % 27; // Wrap around A-Z and '-'
    if (initial == 1){
      initials1[charPos] = letters[letterIndex];
      tft.setCursor(0, score1_cursorY); // Move cursor
      tft.fillRect(0, score1_cursorY, 150, 25, HX8357_BLACK); // Adjust
width/height as needed
      tft.setCursor(0, score1_cursorY); // Move cursor back
      tft.print("1: ");
      tft.println(initials1);
      // tft.fillRect(letterUnderlineX, letterUnderlineY, 20, 3,
HX8357_WHITE); // White underline
      Serial.print("Updated Initials: ");
      Serial.println(initials1);
    }
    else if (initial == 2){
```

```
      initials2[charPos] = letters[letterIndex];
      tft.setCursor(0, score2_cursorY); // Move cursor
      tft.fillRect(0, score2_cursorY, 150, 25, HX8357_BLACK); // Adjust
width/height as needed
      tft.setCursor(0, score2_cursorY); // Move cursor back
      tft.print("2: ");
      tft.println(initials2);
      Serial.print("Updated Initials: ");
      Serial.println(initials2);
    }
    else if (initial == 3){
      initials3[charPos] = letters[letterIndex];
      tft.setCursor(0, score3_cursorY); // Move cursor
      tft.fillRect(0, score3_cursorY, 150, 25, HX8357_BLACK); // Adjust
width/height as needed
      tft.setCursor(0, score3_cursorY); // Move cursor back
      tft.print("3: ");
      tft.println(initials3);
      Serial.print("Updated Initials: ");
      Serial.println(initials3);
    }
  }
}

// Button 2 - Move to next letter
void handleButton2Press(int initial) {
  if (currentState == EDITING) {
    // Erase the previous underline
    if (initial == 1) {
      tft.fillRect(50, letterUnderlineY, 60, 3, HX8357_BLACK); // Clears
all 3 positions
    } else if (initial == 2) {
      tft.fillRect(50, letterUnderlineY, 60, 3, HX8357_BLACK);
    } else if (initial == 3) {
      tft.fillRect(50, letterUnderlineY, 60, 3, HX8357_BLACK); // Fix
incorrect score reference
    }

    charPos++; // Move to next letter
```

```
    if (charPos > 2) {
      Serial.println("Initials Set! Returning to IDLE mode.");
      currentState = GAMEPLAY; // Exit editing mode
      charPos = 0;
    } else {
      letterIndex = 0; // Reset letter choice

      // Update the underline position for the next character
      letterUnderlineX = 50 + (charPos * 20);

      // Draw the new white underline
      tft.fillRect(letterUnderlineX, letterUnderlineY, 20, 3,
HX8357_WHITE);
    }
  }
}

// tasks
void IRAM_ATTR targetISR() {
  targetHit = true;  // Set flag when target is hit
  digitalWrite(LED_BUILTIN,HIGH);
  // Serial.println("DETECTED");
}

void spinnerMotorTask(void *parameter) {
  while (true) {
    if(currentState == GAMEPLAY){
      digitalWrite(enableSPINNER, LOW);
      SPINNER.moveRelativeInSteps(150);  // Always move forward
      digitalWrite(enableSPINNER, HIGH);
    }
    vTaskDelay(100 / portTICK_PERIOD_MS);  // Adjust delay for speed
  }
}

void hoopMotorTask(void *parameter) {
  bool direction = true;

  while (true) {
    if(currentState == GAMEPLAY){
```

```
      digitalWrite(enableHOOP, LOW);

      if (direction) {
        HOOP.moveRelativeInSteps(100);
      } else {
        HOOP.moveRelativeInSteps(-100);
      }

      digitalWrite(enableHOOP, HIGH);
      direction = !direction;
    }

    vTaskDelay(200 / portTICK_PERIOD_MS);  // Wait 200ms between changes
  }
}

void gameplayTask(void *pvParameters) {
  unsigned long currentTime = millis();

  while(1){
  // ChangePalettePeriodically();
  //switchLights();

  currentTime = millis();

  if (targetHit) {  // Check if the target was hit
    targetHit = false;  // Reset flag
    currentScore += 100;
    Serial.printf("Current Score: %d\n", currentScore);
    //play_MONO_wav_file("/Point.wav");
    //size_t bytes_written;
    // updates 7 segment
    currentLightMode = LIGHT_GREEN_BLINK;
    matrix.println(currentScore);
    matrix.writeDisplay();
  }

  if ((currentState == IDLE) || (currentState == EDITING)) {
    highScore();
  }
```

```
// Fixing the GAMEPLAY loop
if (currentState == GAMEPLAY) {
  digitalWrite(LED_BUILTIN, HIGH);

  // Graphics logic
  if (graphicsCounter == 1) {
    testLines(HX8357_CYAN);
  } else if (graphicsCounter == 2) {
    testRects(HX8357_GREEN);
  } else if (graphicsCounter == 3) {
    tft.fillScreen(HX8357_BLACK);
    testCircles(10, HX8357_RED);
  } else if (graphicsCounter == 4) {
    testTriangles();
  } else if (graphicsCounter == 5) {
    //testFilledTriangles();
    graphicsCounter++;
  } else if (graphicsCounter == 6) {
    testRoundRects();
  } else if (graphicsCounter == 7) {
    //testFilledRoundRects();
    graphicsCounter = 0;
  }

  graphicsCounter++;
}

bool sensorState = digitalRead(IRPin1); // Read IR sensor

// Detect ball passing through
if (sensorState == LOW && lastIRState == HIGH) {
  Serial.println("🔴 Ball In Holding Area (Beam Broken)");
  currentState = IDLE;
  currentLightMode = LIGHT_CRAZY;
  Serial.println("Entering IDLE Mode");
  tft.fillScreen(HX8357_BLACK);
  digitalWrite(LED_BUILTIN, LOW);
  //play_MONO_wav_file("/YouLose.wav");
  //size_t bytes_written;
```

```
}
// Detect when the beam is restored
else if (sensorState == HIGH && lastIRState == LOW) {
  Serial.println("✅ Beam Restored (Ball Cleared)");
  currentState = GAMEPLAY;
  currentScore = 0;

  // updates 7 segment
  matrix.println(currentScore);
  matrix.writeDisplay();

  Serial.println("Entering GAMEPLAY Mode");
  tft.fillScreen(HX8357_BLACK);
  digitalWrite(LED_BUILTIN, HIGH);
  graphicsCounter = 1; // Reset graphics counter
}

lastIRState = sensorState;  // Update last state

// Read button states
bool reading1 = digitalRead(buttonPin1);
bool reading2 = digitalRead(buttonPin2);

// Debounce Button 1
if (reading1 != lastButton1State) {
  lastDebounceTime1 = currentTime;
}
if ((currentTime - lastDebounceTime1) > debounceDelay) {
  if ((reading1 != button1State) && (currentState == EDITING)) {
    button1State = reading1;
    while (button1State == LOW) {
      handleButton1Press(scoreToEdit);
      Serial.println("Button1_Pressed");
      delay(250);
      button1State = digitalRead(buttonPin1);
    }
  }
}

// Debounce Button 2
```

```
if (reading2 != lastButton2State) {
  lastDebounceTime2 = currentTime;
}
if ((currentTime - lastDebounceTime2) > debounceDelay) {
  if (reading2 != button2State) {
    button2State = reading2;
    if (button2State == LOW) {
      handleButton2Press(1);
      Serial.println("Button2_Pressed");
    }
  }
}

// Cooldown period to prevent multiple state transitions
const unsigned long COOLDOWN_DELAY = 1000; // 1 second cooldown
static unsigned long lastStateChangeTime = 0;

if (currentState == IDLE) {
  Serial.println("In IDLE mode");

  // Check if editing should be triggered
  if (currentScore > score1) {
    Serial.println("Entering Initials Editing Mode...");
    currentState = EDITING;
    scoreToEdit = 1;
    strcpy(initials3, initials2);
    strcpy(initials2, initials1);
    strcpy(initials1, "AAA");
    score3 = score2;
    score2 = score1;
    score1 = currentScore;
    currentScore = 0;

    // Draw the white underline under the first letter of initials1
    letterUnderlineY = score1_cursorY + 25;
    tft.fillRect(50, letterUnderlineY, 20, 3, HX8357_WHITE);
  } else if (currentScore > score2) {
    Serial.println("Entering Initials Editing Mode...");
    currentState = EDITING;
    scoreToEdit = 2;
```

```
      strcpy(initials3, initials2);
      strcpy(initials2, "AAA");
      score3 = score2;
      score2 = currentScore;
      currentScore = 0;
      // Draw the white underline under the first letter of initials1
      letterUnderlineY = score2_cursorY + 25;
      tft.fillRect(50, letterUnderlineY, 20, 3, HX8357_WHITE);
    } else if (currentScore > score3) {
      Serial.println("Entering Initials Editing Mode...");
      currentState = EDITING;
      strcpy(initials3, "AAA");
      scoreToEdit = 3;
      score3 = currentScore;
      currentScore = 0;
      // Draw the white underline under the first letter of initials1
      letterUnderlineY = score3_cursorY + 25;
      tft.fillRect(50, letterUnderlineY, 20, 3, HX8357_WHITE);
    }

    highScore();
  }

  // Save button states for next loop
  lastButton1State = reading1;
  lastButton2State = reading2;

  vTaskDelay(pdMS_TO_TICKS(5));

}
}

void setup() {
  Serial.begin(115200); // 115200
  delay(1000);
  Serial.println("HX8357D Test!");

  pinMode (LED_BUILTIN, OUTPUT);   // pin 3
  pinMode(buttonPin1, INPUT_PULLUP);
  pinMode(buttonPin2, INPUT_PULLUP);
```

```
pinMode(targetPin1, INPUT_PULLUP);
pinMode(targetPin2, INPUT_PULLUP);

pinMode(IRPin1, INPUT_PULLUP);

attachInterrupt(digitalPinToInterrupt(targetPin1), targetISR, RISING);
attachInterrupt(digitalPinToInterrupt(targetPin2), targetISR, RISING);

// begin lighting
SetupLights();

// begin 7 segment
Wire.begin(SDA, SCL);  // Custom I2C pins
matrix.begin(0x70);
matrix.println(currentScore);
matrix.writeDisplay();

// begin LCD display
tft.begin(8000000);

// read diagnostics (optional but can help debug problems)
uint8_t x = tft.readcommand8(HX8357_RDPOWMODE);
Serial.print("Display Power Mode: 0x"); Serial.println(x, HEX);
x = tft.readcommand8(HX8357_RDMADCTL);
Serial.print("MADCTL Mode: 0x"); Serial.println(x, HEX);
x = tft.readcommand8(HX8357_RDCOLMOD);
Serial.print("Pixel Format: 0x"); Serial.println(x, HEX);
x = tft.readcommand8(HX8357_RDDIM);
Serial.print("Image Format: 0x"); Serial.println(x, HEX);
x = tft.readcommand8(HX8357_RDDSDR);
Serial.print("Self Diagnostic: 0x"); Serial.println(x, HEX);

Serial.println(F("Benchmark                Time (microseconds)"));

tft.fillScreen(HX8357_BLACK);
tft.setRotation(3);
highScore();

Serial.println(F("Done!"));
```

```
  Serial.println("Motors check begin");
  // begin motors
  MotorsSetup();
  spinner.connectToPins(SPINNER_STEP_PIN, SPINNER_DIRECTION_PIN);
  hoop.connectToPins(HOOP_STEP_PIN, HOOP_DIRECTION_PIN);

  // Start as service on core 1 (valid cores: 0 or 1 for ESP32/ESP32S3)
  hoop.startAsService(0);

  // Set initial direction to home (-1 or 1)
  hoop.setDirectionToHome(-1);

  // Example: set a relative target
  hoop.setTargetPositionRelativeInRevolutions(3.0);  // non-blocking
  Serial.println("Motors check end");

  currentState = GAMEPLAY;

  xTaskCreatePinnedToCore(lightTask, "LightTask", 4096, NULL, 1, NULL, 1);
  xTaskCreatePinnedToCore(gameplayTask, "GameplayTask", 4096, NULL, 1,
NULL, 1);
  xTaskCreatePinnedToCore(hoopMotorTask,"HoopMotorTask",2048,NULL,1,NULL,0
);
  xTaskCreatePinnedToCore(spinnerMotorTask, "SpinnerMotorTask", 2048,
NULL, 1, NULL, 1 );
 }

 void loop() {
}
```

lighting.h:

```
#ifndef LIGHTING_H
#define LIGHTING_H

#include <FastLED.h>
#include "pins_arduino.h"
```

```
// Timer group and timer number configuration
#define TIMER_GROUP TIMER_GROUP_0
#define TIMER_NUM TIMER_0

#define NUM_LEDS 120
#define BRIGHTNESS  64
#define UPDATES_PER_SECOND 100
#define LED_TYPE    APA102
#define COLOR_ORDER BGR
CRGB leds[NUM_LEDS];

CRGBPalette16 currentPalette;
TBlendType    currentBlending;

extern CRGBPalette16 myRedWhiteBluePalette;
extern const TProgmemPalette16 myRedWhiteBluePalette_p PROGMEM;

void SetupLights(){
  FastLED.addLeds<APA102, LightingSDA, LightingSCL, RGB>(leds, NUM_LEDS);
// APA102 setup
  // FastLED.addLeds<WS2812, DATA_PIN, RGB>(leds, NUM_LEDS); // Uncomment
for WS2812
  FastLED.setBrightness(  BRIGHTNESS );

  pinMode (LightingSDA, OUTPUT);
  pinMode (LightingSCL, OUTPUT);

  currentPalette = RainbowColors_p;
  currentBlending = LINEARBLEND;
}

void FillLEDsFromPaletteColors( uint8_t colorIndex)
{
    uint8_t brightness = 255;

    for( int i = 0; i < NUM_LEDS; i++) {
        leds[i] = ColorFromPalette( currentPalette, colorIndex,
brightness, currentBlending);
        colorIndex += 3;
    }
```

```
}

// This function fills the palette with totally random colors.
void SetupTotallyRandomPalette()
{
    for( int i = 0; i < 16; i++) {
        currentPalette[i] = CHSV( random8(), 255, random8());
    }
}

// This function sets up a palette of black and white stripes,
// using code.  Since the palette is effectively an array of
// sixteen CRGB colors, the various fill_* functions can be used
// to set them up.
void SetupBlackAndWhiteStripedPalette()
{
    // 'black out' all 16 palette entries...
    fill_solid( currentPalette, 16, CRGB::Black);
    // and set every fourth one to white.
    currentPalette[0] = CRGB::White;
    currentPalette[4] = CRGB::White;
    currentPalette[8] = CRGB::White;
    currentPalette[12] = CRGB::White;

}

// This function sets up a palette of purple and green stripes.
void SetupPurpleAndGreenPalette()
{
    CRGB purple = CHSV( HUE_PURPLE, 255, 255);
    CRGB green  = CHSV( HUE_GREEN, 255, 255);
    CRGB black  = CRGB::Black;

    currentPalette = CRGBPalette16(
                                   green,  green,  black,  black,
                                   purple, purple, black,  black,
                                   green,  green,  black,  black,
                                   purple, purple, black,  black );
}
```

```
// This example shows how to set up a static color palette
// which is stored in PROGMEM (flash), which is almost always more
// plentiful than RAM.  A static PROGMEM palette like this
// takes up 64 bytes of flash.
const TProgmemPalette16 myRedWhiteBluePalette_p PROGMEM =
{
    CRGB::Red,
    CRGB::Gray, // 'white' is too bright compared to red and blue
    CRGB::Blue,
    CRGB::Black,

    CRGB::Red,
    CRGB::Gray,
    CRGB::Blue,
    CRGB::Black,

    CRGB::Red,
    CRGB::Red,
    CRGB::Gray,
    CRGB::Gray,
    CRGB::Blue,
    CRGB::Blue,
    CRGB::Black,
    CRGB::Black
};

void ChangePalettePeriodically()
{
    uint8_t secondHand = (millis() / 1000) % 60;
    static uint8_t lastSecond = 99;

    if( lastSecond != secondHand) {
        lastSecond = secondHand;
        if( secondHand ==  0)  { currentPalette = RainbowColors_p;
currentBlending = LINEARBLEND; }
        if( secondHand == 10)  { currentPalette = RainbowStripeColors_p;
currentBlending = NOBLEND;  }
        if( secondHand == 15)  { currentPalette = RainbowStripeColors_p;
currentBlending = LINEARBLEND; }
```

```
        if( secondHand == 20)  { SetupPurpleAndGreenPalette();
currentBlending = LINEARBLEND; }
        if( secondHand == 25)  { SetupTotallyRandomPalette();
currentBlending = LINEARBLEND; }
        if( secondHand == 30)  { SetupBlackAndWhiteStripedPalette();
currentBlending = NOBLEND; }
        if( secondHand == 35)  { SetupBlackAndWhiteStripedPalette();
currentBlending = LINEARBLEND; }
        if( secondHand == 40)  { currentPalette = CloudColors_p;
currentBlending = LINEARBLEND; }
        if( secondHand == 45)  { currentPalette = PartyColors_p;
currentBlending = LINEARBLEND; }
        if( secondHand == 50)  { currentPalette = myRedWhiteBluePalette_p;
currentBlending = NOBLEND;  }
        if( secondHand == 55)  { currentPalette = myRedWhiteBluePalette_p;
currentBlending = LINEARBLEND; }
    }
}

void switchLights(){
    ChangePalettePeriodically();

    static uint8_t startIndex = 0;
    startIndex = startIndex + 1; /* motion speed */

    FillLEDsFromPaletteColors( startIndex);

    FastLED.show();
    FastLED.delay(1000 / UPDATES_PER_SECOND);
}

// Lighting effect states
enum LightEffect {
  LIGHT_IDLE,
  LIGHT_GREEN_BLINK,
  LIGHT_RED_BLINK,
  LIGHT_CRAZY
};

volatile LightEffect currentLightMode = LIGHT_IDLE;
```

```
unsigned long lightEffectStart = 0;
const unsigned long effectDuration = 1000;  // Duration of blink effects
in ms

void lightTask(void *pvParameters) {
  static LightEffect lastMode = LIGHT_IDLE;
  unsigned long now = millis();

  while (true) {
    now = millis();

    // Handle temporary effects
    if (currentLightMode != LIGHT_IDLE) {
      if (currentLightMode != lastMode) {
        lightEffectStart = now;
        lastMode = currentLightMode;
      }

      switch (currentLightMode) {
        case LIGHT_GREEN_BLINK:
          fill_solid(leds, NUM_LEDS, CRGB::Green);
          break;
        case LIGHT_RED_BLINK:
          fill_solid(leds, NUM_LEDS, CRGB::Red);
          break;
        case LIGHT_CRAZY:
          for (int i = 0; i < NUM_LEDS; i++) {
            leds[i] = CHSV(random8(), 255, 255);
          }
          break;
        default:
          break;
      }

      FastLED.show();

      // End effect after duration
      if (now - lightEffectStart > effectDuration) {
        currentLightMode = LIGHT_IDLE;
        lastMode = LIGHT_IDLE;
```

```
        fill_solid(leds, NUM_LEDS, CRGB::Black);
        FastLED.show();
      }

    } else {
      // Idle mode palette animation (optional)
      static uint8_t index = 0;
      fill_rainbow(leds, NUM_LEDS, index++);
      FastLED.show();
      vTaskDelay(pdMS_TO_TICKS(50));
    }

    vTaskDelay(pdMS_TO_TICKS(5));
  }
}

#endif
```

motors.h:

```
#ifndef Motors_H
#define Motors_H

#include <Arduino.h>
#include <ESP_FlexyStepper.h>

// create the stepper motor object
ESP_FlexyStepper HOOP;
ESP_FlexyStepper SPINNER;

int direction = 1;

int hoopSteps = 2;//20;
int spinnerSteps = 2;//40;

void MotorsSetup(){
  pinMode(enableHOOP, OUTPUT);
  pinMode(enableSPINNER, OUTPUT);
```

```
  pinMode(hooplogic1, OUTPUT);
  pinMode(hooplogic2, OUTPUT);
  pinMode(spinnerlogic2, OUTPUT);
  pinMode(spinnerlogic1, OUTPUT);

  digitalWrite(enableSPINNER, HIGH);
  // set to 1 step mode
  digitalWrite(hooplogic1, LOW);
  digitalWrite(hooplogic2, LOW);

  digitalWrite(spinnerlogic1, LOW);
  digitalWrite(spinnerlogic2, LOW);

  // turn off stepper motor
  digitalWrite(enableHOOP, HIGH);
  Serial.begin(115200);
  // connect and configure the stepper motor to its IO pins
  HOOP.connectToPins(HOOP_STEP_PIN, HOOP_DIRECTION_PIN);
  SPINNER.connectToPins(SPINNER_STEP_PIN, SPINNER_DIRECTION_PIN);

  // set the speed and acceleration rates for the stepper motor
  HOOP.setSpeedInStepsPerSecond(500);
  HOOP.setAccelerationInStepsPerSecondPerSecond(100);
  SPINNER.setSpeedInStepsPerSecond(500);
  SPINNER.setAccelerationInStepsPerSecondPerSecond(1000);
}

void turnMotorsForward(){
    // for (int i=0; i<=40; i++){
      // hoop move
      digitalWrite(enableHOOP, LOW);
      HOOP.moveRelativeInSteps(hoopSteps); // 5
      digitalWrite(enableHOOP, HIGH);
      // spinner move

      digitalWrite(enableSPINNER, LOW);
      SPINNER.moveRelativeInSteps(spinnerSteps);  // 10
      digitalWrite(enableSPINNER, HIGH);
    // }
}
```

```
void turnMotorsBackward(){
    // rotate backward 1 rotation, then wait 1 second
    // for (int i=0; i<=40; i++){
      // hoop move backward
      digitalWrite(enableHOOP, LOW);
      HOOP.moveRelativeInSteps(-1*hoopSteps); // -5
      digitalWrite(enableHOOP, HIGH);
      // spinner move
      digitalWrite(enableSPINNER, LOW);
      SPINNER.moveRelativeInSteps(spinnerSteps); // 10
      digitalWrite(enableSPINNER, HIGH);
  // }
}

#endif // Motors_H
```

audio.h:

```
#ifndef AUDIO_H
#define AUDIO_H

#include <math.h>
#include "driver/i2s_std.h"
#include "driver/gpio.h"
#include "FS.h"
#include "SPIFFS.h"

#define SAMPLE_RATE 48000
#define FREQUENCY 1000  // 1 kHz sine wave
#define AMPLITUDE (2147483647 / 4)  // Reduce amplitude to 25% max
#define BUFFER_SIZE 256  // Number of stereo frames per buffer


static const char *TAG = "WAV_PLAYER";


int buttonState;
int i = 1;
```

```
static int32_t audio_buffer[BUFFER_SIZE];  // Mono buffer


i2s_chan_handle_t tx_handle;
i2s_chan_config_t chan_cfg = I2S_CHANNEL_DEFAULT_CONFIG(I2S_NUM_AUTO,
I2S_ROLE_MASTER);
i2s_std_config_t std_cfg = {
    .clk_cfg = I2S_STD_CLK_DEFAULT_CONFIG(SAMPLE_RATE),
    .slot_cfg = I2S_STD_MSB_SLOT_DEFAULT_CONFIG(I2S_DATA_BIT_WIDTH_32BIT,
I2S_SLOT_MODE_MONO),  // Set MONO mode
    .gpio_cfg = {
        .mclk = I2S_GPIO_UNUSED,
        .bclk = GPIO_NUM_15,
        .ws = GPIO_NUM_3, //7, //21,
        .dout = GPIO_NUM_16, // GPIO_NUM_47,
        .din = I2S_GPIO_UNUSED,
        .invert_flags = {
            .mclk_inv = false,
            .bclk_inv = false,
            .ws_inv = false,
        },
    },
};


void generate_MONO_sine_wave() {
  static float phase = 0.0f;
  float phase_increment = 2.0f * M_PI * FREQUENCY / SAMPLE_RATE;

  for (int i = 0; i < BUFFER_SIZE; i++) {
      audio_buffer[i] = (int32_t)(AMPLITUDE * sinf(phase));  // Single
channel (mono)
      phase += phase_increment;
      if (phase >= 2.0f * M_PI) phase -= 2.0f * M_PI;
  }
}


void play_MONO_wav_file(const char *file_path) {
```

```
    i2s_channel_enable(tx_handle);

    File file = SPIFFS.open(file_path, "rb");  // Open file using SPIFFS
    if (!file) {
        Serial.printf("Failed to open file: %s\n", file_path);
        return;
    }

    // Read the WAV header
    uint8_t header[44];
    if (file.read(header, 44) != 44) {  // Ensure we actually read 44
bytes
        Serial.println("Failed to read WAV header");
        file.close();
        return;
    }

    // Check if the file is a valid WAV file
    if (memcmp(header, "RIFF", 4) != 0 || memcmp(header + 8, "WAVE", 4) !=
0) {
        Serial.println("Invalid WAV file");
        file.close();
        return;
    }

    // Extract audio data information
    uint32_t sample_rate = *(uint32_t *)&header[24];
    uint16_t num_channels = *(uint16_t *)&header[22];
    uint16_t bit_depth = *(uint16_t *)&header[34];

    Serial.printf("Sample Rate: %d, Channels: %d, Bit Depth: %d\n",
sample_rate, num_channels, bit_depth);

    // Adjust I2S configuration for MONO
    std_cfg.clk_cfg.sample_rate_hz = sample_rate;
    std_cfg.slot_cfg.slot_mode = I2S_SLOT_MODE_MONO;
    std_cfg.slot_cfg.data_bit_width = (bit_depth == 16) ?
I2S_DATA_BIT_WIDTH_16BIT : I2S_DATA_BIT_WIDTH_32BIT;

    int16_t audio_buffer[BUFFER_SIZE];  // Mono buffer
```

```
    size_t bytes_read, bytes_written;

    while ((bytes_read = file.read((uint8_t*)audio_buffer,
sizeof(audio_buffer))) > 0) {
        esp_err_t err = i2s_channel_write(tx_handle, (void*)audio_buffer,
bytes_read, &bytes_written, pdMS_TO_TICKS(1000));
        if (err != ESP_OK) {
            Serial.printf("I2S write error: %d\n", err);
            break;
        }
    }

    file.close();  // Ensure file is closed
    Serial.println("Finished playing WAV file");

    // Stop I2S output to prevent noise after playback
    i2s_channel_disable(tx_handle);
}


#endif // AUDIO_H
```

LCD_Graphics.h

```
#ifndef _LCD_Graphics_H
#define _LCD_Graphics_H

#include <Adafruit_HX8357.h>   // Include required library
#include <Arduino.h>            // For `micros()` function
#define HX8357_ORANGE 0xFD20  // RGB565 format

// Declare the display object externally (to be initialized in main code)
extern Adafruit_HX8357 tft;

// Ball animation state
extern int ballX, ballY;
extern int ballDX, ballDY;
extern uint8_t ballHue;
```

```
extern unsigned long lastBallUpdate;
extern int ballRadius;

void initPinballScene() {
  tft.fillScreen(HX8357_BLACK);

  // Static background (only drawn once)
  tft.setTextColor(HX8357_YELLOW);
  tft.setTextSize(3);
  tft.setCursor(40, 20);
  tft.print("PINBALL");

  tft.fillCircle(60, 100, 15, HX8357_RED);
  tft.fillCircle(160, 100, 15, HX8357_BLUE);
  tft.fillCircle(110, 160, 15, HX8357_GREEN);

  tft.fillRect(45, 200, 40, 10, HX8357_WHITE);
  tft.fillRect(135, 200, 40, 10, HX8357_WHITE);
  tft.fillTriangle(85, 200, 110, 220, 135, 200, HX8357_CYAN);

  tft.drawRect(20, 250, 180, 30, HX8357_RED);
  tft.setCursor(30, 260);
  tft.setTextColor(HX8357_WHITE);
  tft.setTextSize(2);
  tft.print("Score: 000000");

  tft.setTextColor(HX8357_MAGENTA);
  tft.setTextSize(1);
  tft.setCursor(65, 290);
  tft.print("Press Any Button to Start");
}

void updateBallPosition() {
  unsigned long now = millis();
  if (now - lastBallUpdate < 30) return;  // 30ms throttle for ~33 FPS
  lastBallUpdate = now;

  // Erase previous ball
  tft.fillCircle(ballX, ballY, ballRadius, HX8357_BLACK);
```

```
  // Update ball position
  ballX += ballDX;
  ballY += ballDY;

  // Bounce off edges
  if (ballX - ballRadius < 0 || ballX + ballRadius > tft.width()) ballDX =
-ballDX;
  if (ballY - ballRadius < 0 || ballY + ballRadius > tft.height()) ballDY
= -ballDY;

  // Redraw ball at new position
  tft.fillCircle(ballX, ballY, ballRadius, HX8357_WHITE);
}


String lcdMessage = "";
unsigned long lcdMessageTime = 0;
const unsigned long lcdMessageDuration = 1000; // 1 second

unsigned long testLines(uint16_t color) {
    unsigned long start = micros();
    int x1 = 0, y1 = 0, x2, y2;
    int w = tft.width();
    int h = tft.height();

    tft.fillScreen(HX8357_BLACK);

    y2 = h - 1;
    for (x2 = 0; x2 < w; x2 += 6) tft.drawLine(x1, y1, x2, y2, color);
    x2 = w - 1;
    for (y2 = 0; y2 < h; y2 += 6) tft.drawLine(x1, y1, x2, y2, color);

    return micros() - start;
}

unsigned long testFastLines(uint16_t color1, uint16_t color2) {
    unsigned long start = micros();
    int x, y, w = tft.width(), h = tft.height();

    tft.fillScreen(HX8357_BLACK);
```

```
    for (y = 0; y < h; y += 5) tft.drawFastHLine(0, y, w, color1);
    for (x = 0; x < w; x += 5) tft.drawFastVLine(x, 0, h, color2);

    return micros() - start;
}

unsigned long testRects(uint16_t color) {
    unsigned long start = micros();
    int n, i, i2, cx = tft.width() / 2, cy = tft.height() / 2;

    tft.fillScreen(HX8357_BLACK);
    n = min(tft.width(), tft.height());

    for (i = 2; i < n; i += 6) {
        i2 = i / 2;
        tft.drawRect(cx - i2, cy - i2, i, i, color);
    }

    return micros() - start;
}

unsigned long testFilledRects(uint16_t color1, uint16_t color2) {
    unsigned long start, t = 0;
    int n, i, i2, cx = tft.width() / 2 - 1, cy = tft.height() / 2 - 1;

    tft.fillScreen(HX8357_BLACK);
    n = min(tft.width(), tft.height());

    for (i = n; i > 0; i -= 6) {
        i2 = i / 2;
        start = micros();
        tft.fillRect(cx - i2, cy - i2, i, i, color1);
        t += micros() - start;
        tft.drawRect(cx - i2, cy - i2, i, i, color2);
    }

    return t;
}
```

```
unsigned long testFilledCircles(uint8_t radius, uint16_t color) {
    unsigned long start = micros();
    int x, y, w = tft.width(), h = tft.height(), r2 = radius * 2;

    tft.fillScreen(HX8357_BLACK);

    for (x = radius; x < w; x += r2) {
        for (y = radius; y < h; y += r2) {
            tft.fillCircle(x, y, radius, color);
        }
    }

    return micros() - start;
}

unsigned long testCircles(uint8_t radius, uint16_t color) {
    unsigned long start = micros();
    int x, y, r2 = radius * 2, w = tft.width() + radius, h = tft.height()
+ radius;

    for (x = 0; x < w; x += r2) {
        for (y = 0; y < h; y += r2) {
            tft.drawCircle(x, y, radius, color);
        }
    }

    return micros() - start;
}

unsigned long testTriangles() {
    unsigned long start = micros();
    int n, i, cx = tft.width() / 2 - 1, cy = tft.height() / 2 - 1;

    tft.fillScreen(HX8357_BLACK);
    n = min(cx, cy);

    for (i = 0; i < n; i += 5) {
        tft.drawTriangle(
            cx, cy - i,          // peak
            cx - i, cy + i,      // bottom left
```

```
            cx + i, cy + i,      // bottom right
            tft.color565(200, 20, i)
        );
    }

    return micros() - start;
}

unsigned long testFilledTriangles() {
    unsigned long start, t = 0;
    int i, cx = tft.width() / 2 - 1, cy = tft.height() / 2 - 1;

    tft.fillScreen(HX8357_BLACK);

    for (i = min(cx, cy); i > 10; i -= 5) {
        start = micros();
        tft.fillTriangle(
            cx, cy - i,
            cx - i, cy + i,
            cx + i, cy + i,
            tft.color565(0, i, i)
        );
        t += micros() - start;
        tft.drawTriangle(
            cx, cy - i,
            cx - i, cy + i,
            cx + i, cy + i,
            tft.color565(i, i, 0)
        );
    }

    return t;
}

unsigned long testRoundRects() {
    unsigned long start = micros();
    int w, i, i2, cx = tft.width() / 2, cy = tft.height() / 2;

    tft.fillScreen(HX8357_BLACK);
    w = min(tft.width(), tft.height());
```

```
    for (i = 0; i < w; i += 8) {
        i2 = i / 2 - 2;
        tft.drawRoundRect(cx - i2, cy - i2, i, i, i / 8, tft.color565(i,
100, 100));
    }

    return micros() - start;
}

unsigned long testFilledRoundRects() {
    unsigned long start;
    int           i, i2,
                  cx = tft.width()  / 2 + 10,
                  cy = tft.height() / 2 + 10;

    tft.fillScreen(HX8357_BLACK);
    start = micros();
    for(i=min(tft.width(), tft.height()) - 20; i>25; i-=6) {
      i2 = i / 2;
      tft.fillRoundRect(cx-i2, cy-i2, i-20, i-20, i/8, tft.color565(100,
i/2, 100));
    }

    return micros() - start;
  }

void flashPoints(const char* message, uint16_t color) {
  tft.fillScreen(HX8357_BLACK);
  tft.setTextSize(5);
  tft.setTextColor(color);

  int y = 80;
  for (int i = 0; i < 5; i++) { // Reduced from 10 to 5 steps
    tft.fillScreen(HX8357_BLACK);  // Clear for clean animation
    tft.setCursor(60, y + i * 6);  // Move down slightly per frame
    tft.print(message);
    delay(60); // Faster transitions
  }
```

```
  tft.fillScreen(HX8357_BLACK); // Final clear after animation
}



void drawPinballSplashHX8357() {
  static int lastBallX = ballX;
  static int lastBallY = ballY;

  // Clear the previous ball (but not full screen to avoid flicker)
  tft.fillCircle(lastBallX, lastBallY, 6, HX8357_BLACK);

  // Draw static parts once (optional optimization)
  static bool initialized = false;
  if (!initialized) {
    tft.fillScreen(HX8357_BLACK);
    tft.setTextColor(HX8357_YELLOW);
    tft.setTextSize(3);
    tft.setCursor(60, 10);
    tft.print("PINBALL");

    tft.fillCircle(60, 80, 15, HX8357_RED);
    tft.fillCircle(160, 80, 15, HX8357_BLUE);
    tft.fillCircle(110, 130, 15, HX8357_GREEN);

    tft.fillRect(45, 200, 40, 10, HX8357_WHITE);
    tft.fillRect(135, 200, 40, 10, HX8357_WHITE);
    tft.fillTriangle(85, 200, 110, 220, 135, 200, HX8357_CYAN);

    tft.drawRect(20, 250, 180, 30, HX8357_RED);
    tft.setCursor(30, 260);
    tft.setTextColor(HX8357_WHITE);
    tft.setTextSize(2);
    tft.print("Score: 000000");

    initialized = true;
  }

  // Update ball hue and position
  ballHue += 5;
```

```
  lastBallX = ballX;
  lastBallY = ballY;
  ballX += ballDX;
  ballY += ballDY;

  // Bounce within screen bounds (adjust to your playfield size)
  if (ballX <= 20 || ballX >= 200) ballDX = -ballDX;
  if (ballY <= 30 || ballY >= 240) ballDY = -ballDY;

  // Convert HSV to RGB565 color
  uint16_t ballColor = tft.color565(
    (uint8_t)(sin(ballHue * 0.1) * 127 + 128),
    (uint8_t)(cos(ballHue * 0.1) * 127 + 128),
    (uint8_t)(sin(ballHue * 0.15 + 1) * 127 + 128)
  );

  // Draw new ball
  tft.fillCircle(ballX, ballY, 6, ballColor);
}




void showGameOverScreen() {
  tft.fillScreen(HX8357_BLACK);
  tft.setTextSize(6);
  tft.setTextColor(HX8357_RED);

  String msg = "GAME OVER";

  int16_t x1, y1;
  uint16_t w, h;
  tft.getTextBounds(msg, 0, 0, &x1, &y1, &w, &h);

  int x = (tft.width() - w) / 2;
  int y = (tft.height() - h) / 2;

  tft.setCursor(x, y);
  tft.print(msg);
```

```
  delay(2000);  // Display for 2 seconds
  tft.fillScreen(HX8357_BLACK);  // Clear before showing high scores
}




#endif  // _LCD_Graphics_H
```

SPIFFS_Tasks.h

```
#ifndef SPIFFS_TASKS_H
#define SPIFFS_TASKS_H

#include "FS.h"
#include "SPIFFS.h"

void listDir(fs::FS &fs, const char *dirname, uint8_t levels) {
    Serial.printf("Listing directory: %s\r\n", dirname);
    File root = fs.open(dirname);
    if (!root) {
        Serial.println("- failed to open directory");
        return;
    }
    if (!root.isDirectory()) {
        Serial.println(" - not a directory");
        return;
    }

    File file = root.openNextFile();
    while (file) {
        if (file.isDirectory()) {
            Serial.print("  DIR : ");
            Serial.println(file.name());
            if (levels) {
                listDir(fs, file.name(), levels - 1);
            }
        } else {
            Serial.print("  FILE: ");
```

```
            Serial.print(file.name());
            Serial.print("\tSIZE: ");
            Serial.println(file.size());
        }
        file = root.openNextFile();
    }
}

void readFile(fs::FS &fs, const char *path) {
    Serial.printf("Reading file: %s\r\n", path);
    File file = fs.open(path);
    if (!file || file.isDirectory()) {
        Serial.println("- failed to open file for reading");
        return;
    }

    Serial.println("- read from file:");
    while (file.available()) {
        Serial.write(file.read());
    }
    file.close();
}

void writeFile(fs::FS &fs, const char *path, const char *message) {
    Serial.printf("Writing file: %s\r\n", path);
    File file = fs.open(path, FILE_WRITE);
    if (!file) {
        Serial.println("- failed to open file for writing");
        return;
    }

    if (file.print(message)) {
        Serial.println("- file written");
    } else {
        Serial.println("- write failed");
    }
    file.close();
}

void appendFile(fs::FS &fs, const char *path, const char *message) {
```

```
    Serial.printf("Appending to file: %s\r\n", path);
    File file = fs.open(path, FILE_APPEND);
    if (!file) {
        Serial.println("- failed to open file for appending");
        return;
    }

    if (file.print(message)) {
        Serial.println("- message appended");
    } else {
        Serial.println("- append failed");
    }
    file.close();
}

void renameFile(fs::FS &fs, const char *path1, const char *path2) {
    Serial.printf("Renaming file %s to %s\r\n", path1, path2);
    if (fs.rename(path1, path2)) {
        Serial.println("- file renamed");
    } else {
        Serial.println("- rename failed");
    }
}

void deleteFile(fs::FS &fs, const char *path) {
    Serial.printf("Deleting file: %s\r\n", path);
    if (fs.remove(path)) {
        Serial.println("- file deleted");
    } else {
        Serial.println("- delete failed");
    }
}

void testFileIO(fs::FS &fs, const char *path) {
    Serial.printf("Testing file I/O with %s\r\n", path);
    static uint8_t buf[512];
    size_t len = 0;
    File file = fs.open(path, FILE_WRITE);
    if (!file) {
        Serial.println("- failed to open file for writing");
```

```
        return;
    }

    Serial.print("- writing");
    uint32_t start = millis();
    for (size_t i = 0; i < 2048; i++) {
        if ((i & 0x001F) == 0x001F) {
            Serial.print(".");
        }
        file.write(buf, 512);
    }
    Serial.println("");
    uint32_t end = millis() - start;
    Serial.printf(" - %u bytes written in %u ms\r\n", 2048 * 512, end);
    file.close();

    file = fs.open(path);
    start = millis();
    end = start;
    size_t i = 0;
    if (file && !file.isDirectory()) {
        len = file.size();
        size_t flen = len;
        start = millis();
        Serial.print("- reading");
        while (len) {
            size_t toRead = len > 512 ? 512 : len;
            file.read(buf, toRead);
            if ((i++ & 0x001F) == 0x001F) {
                Serial.print(".");
            }
            len -= toRead;
        }
        Serial.println("");
        end = millis() - start;
        Serial.printf("- %u bytes read in %u ms\r\n", flen, end);
        file.close();
    } else {
        Serial.println("- failed to open file for reading");
    }
```

```
}

void readFile(const char *path) {
    File file = SPIFFS.open(path, "r");
    if (file) {
        Serial.print("File: " + String(path) + ", Content: ");

        if (String(path).endsWith(".bin")) {
            while (file.available()) {
                Serial.print(file.read(), HEX);
                Serial.print(" ");
            }
        } else {
            Serial.print(file.readString());
        }

        Serial.println();
        file.close();
    } else {
        Serial.println("Failed to open file for reading: " +
String(path));
    }
}

void writeBinaryDataToFile(const char *path) {
    Serial.print("Creating file with binary data: ");
    Serial.println(String(path));

    File file = SPIFFS.open(path, "w");
    if (file) {
        uint8_t binaryData[] = {0x01, 0x02, 0x03, 0x04};
        file.write(binaryData, sizeof(binaryData));
        file.close();
        Serial.println("Binary file created and written successfully");
    } else {
        Serial.println("Failed to create binary file");
    }
}

void writeFileWithPrint(const char *path, const char *content) {
```

```
    Serial.print("Creating file with SPIFFS.print(): ");
    Serial.println(String(path));

    File file = SPIFFS.open(path, "w");
    if (file) {
        file.print(content);
        file.close();
        Serial.println("Text file created and written successfully");
    } else {
        Serial.println("Failed to create text file");
    }
}

void readAndPrintFiles() {
    Serial.println("Reading and printing files:");
    readFile("/binaryFile.bin");
    readFile("/textFile.txt");
}

void listFiles(const char *dir) {
    Serial.print("Listing files in directory: ");
    Serial.println(String(dir));

    File root = SPIFFS.open(dir);
    if (!root) {
        Serial.println("Failed to open directory");
        return;
    }

    File file = root.openNextFile();
    while (file) {
        Serial.println("File: " + String(file.name()) + ", Size: " +
String(file.size()));
        file = root.openNextFile();
    }
}

#endif // SPIFFS_TASKS_H
```

Libraries:

Adafruit LED Backpack Library

7 Segment Library

ESP-FlexyStepper Library

Appendix C: Datasheet Links

Relevant parts or component data sheets (do NOT include the data sheets for the microcontroller or other huge files but give good links to where they may be found.)

Data Sheet Links:
5mm IR beam break sensor: The sensing distance is around 50cm (20 inches). The power supply voltage is 3.3V to 5.5V DC, but it works better at the higher end of that range. The emitter current draw is 10 mA at 3.3V and 20 mA at 5V. The receiver's output is open collector, so an enabled pull-up resistor is necessary.

3mm IR beam break sensor: The sensing distance is around 25cm (10 inches), half that of the 5mm IR beams. The power supply voltage is 3.3V to 5.5V DC, but it works better at the higher end of that range. The emitter current draw is 10mA at 3.3V and 20mA at 5V. The receiver's output is open collector, so an enabled pull-up resistor is necessary.

Stepper motor (x2): This bipolar stepper motor has a step angle of 1.8 degrees per step, or 200 steps per revolution.

Microswitch (x2): This switch type is SPDT (Single Pole, Double Throw), which features a common, normally open (NO), and normally closed (NC) terminal. The electrical rating is 5A at 125/250V AC for standard load and 0.1A at 125/250V AC for low-current versions.

Barrel Jack 2.50mm: Jack connector to a barrel plug in with a normally closed, single switch kinked pin. Featuring through hole mounting at a right angle with a 2.50mm inner diameter and 5.50mm outer diameter, while rated up to 24VDC.

Linear Voltage Regulator (5V 3A): Linear voltage regulator IC with a 30V max input to a fixed 5V output. The voltage drops out at 1.5V at 3A.

Linear Voltage Regulator (3.3V 2A): Linear voltage regulator IC with a 6V max input to a fixed 3.3V output. The voltage drops out at 0.25V at 2A.