EE 41440: Senior Design II

Desktop Pinball Machine

Final Report

Alisa Nguyen, Lauren Rymsza, Victoria Smerdon, Yadviga Tischenko, James Venditto

3 May 2022

1 Introduction

Our project is motivated by a desire to create something that can improve the recreational lives of our fellow students at Notre Dame. The University provides students with countless resources for academic and professional development, but when it comes to recreational activities, students struggle to find options that can fit into their busy schedules. Students are overwhelmed by stress from daily academic and extracurricular commitments, and their demanding schedules leave only small gaps of time throughout the day in between classes and meetings that often go to waste, and are too short to participate in longer recreational activities. With online platforms crowding university spaces, students long for a source of "unplugged" entertainment that can be shared with friends. Students need an offline means of recreation that can be used for short breaks of time, played either alone or with friends, and can easily be placed in a college dorm room. It also should be fun and eye-catching to a college-aged audience.

We proposed a pinball machine that can fit on a standard Notre Dame dorm room desktop. Our pinball machine aims to offer a communal source of offline entertainment for students, that can be used for either short or long periods of time, and can easily be moved between dorm rooms. Pinball is an "arcade entertainment machine" where users launch a ball into the playing field. As the ball interacts with the various bumpers and targets, points are obtained. As the ball falls due to the angle of the board, the user can control flippers at the bottom to send the ball in a new direction. Once the ball returns to the gutter, the game is over.

Once our machine is plugged into a wall outlet and the ball placed on the plunger spring, it is ready to play. An ambient pattern flashes on the LED strip to entertain the user and draw in onlookers. The user can use the plunger to launch the ball into the playing field of the pinball machine and begin their game. While the ball is in the playing field, the user presses buttons to control flippers, trying to keep the ball in the playing field. When the ball hits the targets at the top of the playing field, the microcontroller detects the impact and increments the score, which is displayed at the top of the machine on a 7-segment display. Scoring also causes a sound effect to play and a colorful routine to flash on the LED strip. A second type of target, called “slingshots”, push back on the ball when struck, redirecting its path and adding to the difficulty of the game. When the ball passes through the flippers and rolls down the bottom path, it triggers a beam-break infrared sensor when passing back into the gutter, which signals the game is over, clearing the score and activating a sound and LED routine. The user is then free to launch the ball and start a new game.

The main body of the machine is rectangular on an incline of approximately 6.5 degrees with respect to the ground, and the play area is also rectangular with raised features. These features serve as obstacles for the balls, alongside targets that detect a ball strike for score-keeping, as well as the aforementioned slingshots. The microcontroller keeps track of the score in memory, and prints it to the left-hand 7-segment display via I2C communication. A second 7-segment display displays a four-letter message that adds to the ambience; for our demonstration it says “NDEE”. A mechanical plunger located in the side gutter allows the user to launch the ball into the playing field while offering versatility with regard to how hard or softly they want to launch it. The flippers are powered by solenoids, which use a magnetic field to retract a plunger when a voltage is applied through a switch press, which in turn bats the flipper. Two push buttons on either side of the chassis allow the user to control them and strike the ball.

Throughout play, the LED strip and speaker provide a festive atmosphere for the user. As aforementioned, the LED offers both constant ambience and explicitly reacts to scoring events and the end of the game. The speaker plays sound effects during both of the latter events, but possesses the capability to play ambient audio if desired; this feature was removed due to negative user feedback. Both peripherals are controlled via the microcontroller: the speaker by I2S communication performed via an audio amplifier board, and the LED strip by digital switching through power MOSFETS. Audio clips are stored in the microcontroller's memory in the form of header files, and LED routines are functions written into the main program.

The final desktop pinball machine build is shown below in Figure 1.

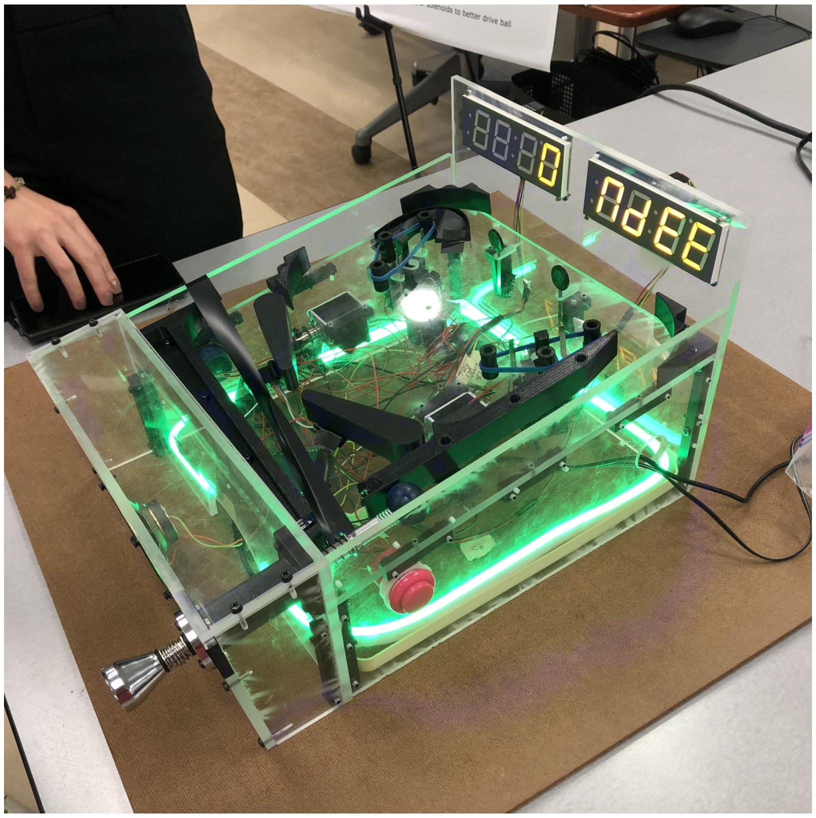


Figure 1. Desktop Pinball Machine Final Build.

Our project generally fulfilled our expectations. It is a complete, playable pinball machine that can fit on a standard desktop and is easily carryable by a student. It lights up, catches the attention of passersby, and reacts accordingly to scoring and the end of the run. However, there are areas where it is not as reliable as intended, and would have to be improved before a consumer model was released. Currently, the ball used is not heavy enough to depress the slingshot or target switches enough to trigger this mechanism, meaning that it is nearly impossible to score points. Additionally, our machine does not possess a full bottom panel, which means that wires occasionally stick out, and it may be slightly more unwieldy to carry (although it is light enough from a weight perspective). Lastly, the bottom of the playing field does not have a steep enough incline for the ball to consistently roll into the gutter; sometimes it becomes stuck.

Although it currently lacks the degree of accuracy or reliability we ultimately want, it does fulfill its system requirements, and these remaining issues are of a mechanical nature, and can be fixed with a relatively simple amount of further iteration, which will be discussed in more detail in Section 6.

2 Detailed System Requirements

The system requirements for each of the major parts of Desktop Pinball Machine are as follows:

Power

- Wall connection: Must be able to connect to wall outlet, rectify wall voltage, and scale down to three necessary power levels
- Current Draw: Must be able to handle currents of several Amps at a time that are drawn by solenoids and LED strip during operation
- Power Level 1 (12 V): Must be able to obtain 12 V and source several Amps from wall supply to power the push-pull solenoids used for flippers and slingshots, power LED strip, and be source for step-down to lower voltage levels
- Power Level 2 (5 V): Must be able to obtain 5 V by scaling down from 12 V wall supply, to be used for 7-segment display and to source step-down to lowest voltage level
- Power Level 3 (3.3 V): Must be able to to obtain 5 V by scaling down from 5 V supply, to be used for audio and all microcontroller logic
- Isolation: Must be able to isolate different voltage and current levels from one another to protect sensitive equipment

Display

- Size: Must have enough rows and columns to display all the information we need - current score with enough place values
- Communication Protocol: Must use a communication protocol compatible with our microcontroller (serial, I2C, or SPI)
- Refresh Rate: Must refresh quickly enough to not miss any score updates

Entertainment/Atmosphere

- Audio
  - Memory: Must have ability to save audio files in memory
  - Number of audio files: Must have a unique sound for at least scoring and game over, with ability to add more if necessary
  - Compatibility: Speaker, amplifier, and DAC must work with microcontroller communication protocols and bit processing as necessary

- Lights
  - Power: LED must either be powerable by microcontroller, or have circuitry to control higher voltages from microcontroller (such as power MOSFET's)
  - Programming: LED should be programmable, with choices of different colors
  - Variety: Should have different lighting routines for different elements of the game, including:
    - Background lighting
    - Game over
    - Score incremented

Active Obstacles

- Electrical Control: Must be capable of being triggered by a ball strike
- Durability: Must be able to withstand repeated strikes from the ball
- Strength: Must impart enough force to redirect the ball

Targets

- Responsiveness: Must react nearly instantaneously to ball striking a scoring element
- Accuracy: Must count each hit once and only once, and not present any "false positives" (reacting when something has not hit the target)
- Durability: Must be able to withstand repeated strikes from the ball

Physical Structure

- Size: Fit on a standard Notre Dame dorm room desktop:
    - 20 ¾ inches x 41 ¾ inches (53 cm x 106 cm)
- Weight: Light enough to be transported by a single college student and sturdy enough to withstand indoor conditions
- Angle: Must be presented at a significant enough angle that the ball will roll down toward the gutter reasonably fast, but not so severe that the flippers can't fight gravity: at approximately 6.5 degrees
- Aesthetic Appeal: Must look presentable for its purpose as an entertainment device

3 Detailed Project Description

3.1 System Theory of Operation

The system works as described in section 1. The user pulls back on the plunger with a ball loaded in the gutter, and releases to mechanically fire the ball into the playing field. Once there, they press the side buttons to complete the solenoid circuit and energize the flippers, firing them; the flippers relax when the buttons are released. As this is happening, the LED strip cycles through a color cycle of green, then blue, then yellow, which is dictated by the main loop of the software and controlled by microcontroller I/O pins. If the ball hits a target, the corresponding limit switch is closed, triggering an interrupt on the connected I/O pin. This interrupt increments the score variable, and signals to the main task that the 7-segment score display be updated with the new score via I2C communication, and a scoring LED routine begins. Simultaneously, it signals for the audio task to play a scoring sound, which is played on the speaker via I2S communication. If a slingshot is struck, the closed limit switch completes the analog solenoid circuit, firing the slingshot to redirect the ball. Although not currently active in our design, it is intended that the fired solenoid depress another limit switch, connected to a microcontroller I/O port, which triggers the same aforementioned scoring interrupt. Finally, if the ball falls to the end of the playing field, it will roll past the beam-break IR sensor, temporarily breaking the beam and triggering an I/O interrupt. This interrupt resets the score variable to 0, signals for the 7-segment scoring display to write this 0 as the new score, begins the “game over” LED routine, and signals for the audio task to play the “game over” audio. With the ball back in the gutter, a new game can be started. The microcontroller is housed in the underside cavity of the chassis, alongside the 12 V protoboard. The model used for this design was the ESP-32-WROOM-32E-N8 module. The complete system block diagram of our project can be found below in figure 2.

3.2 System Block Diagram

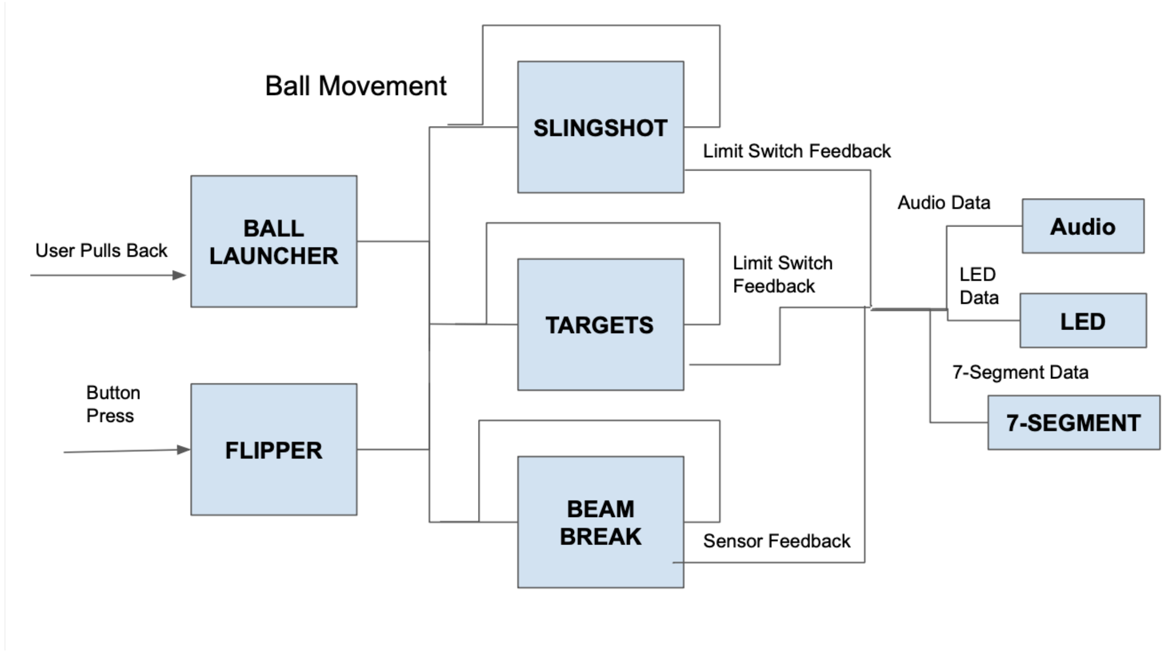


Figure 2. System Block Diagram.

Detailed Operation of Subsystems

3.3 Description of Chassis/Playing Field

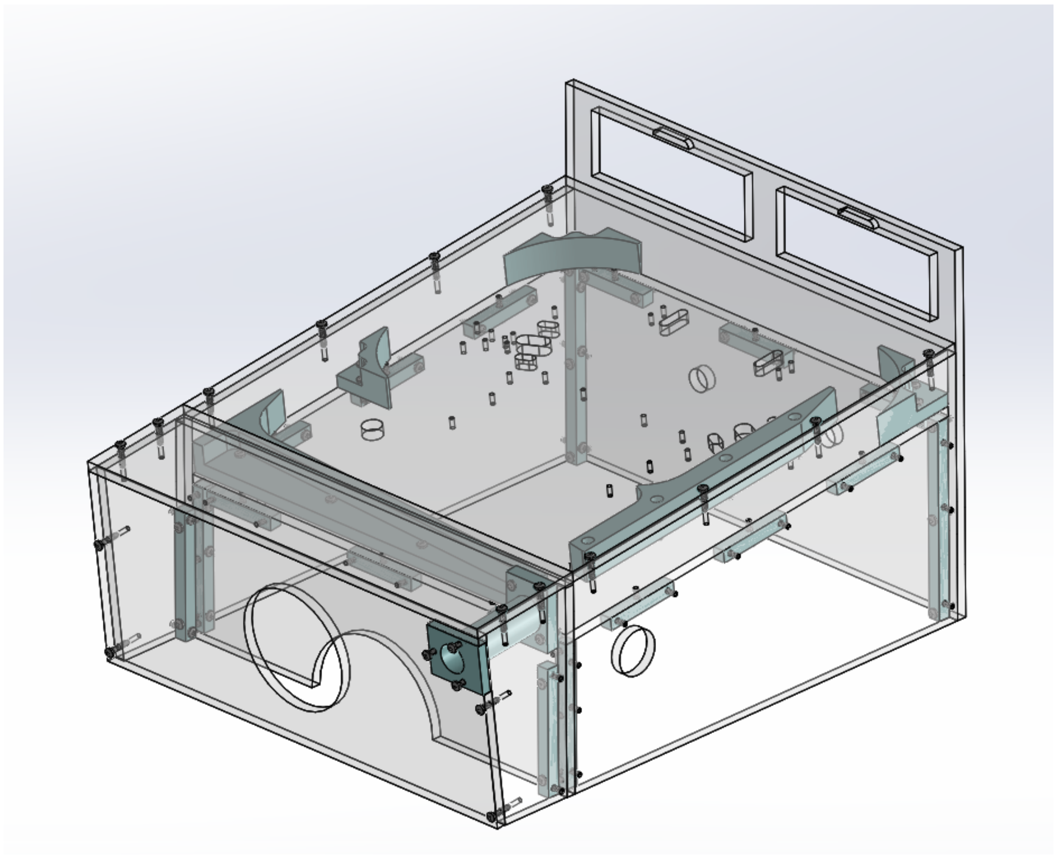
The chassis of the tabletop pinball machine is an intricate mechanical subsystem. The primary material used for the frame of our tabletop pinball machine is ¼ inch thick transparent casted acrylic. A waterjet and Haas Machine at the Engineering Innovation Hub was used to cut the acrylic and create the appropriate holes, according to the coordinates of the holes on the sheet of acrylic as designed in SOLIDWORKS. Each of the holes was tapped with a drill for the insertion of a screw. M3 screws with lengths ranging from 12 mm - 16 mm were used depending on the placement in the build. Additional larger holes were included to allow for the power cables and entry of wires from outside subsystems.

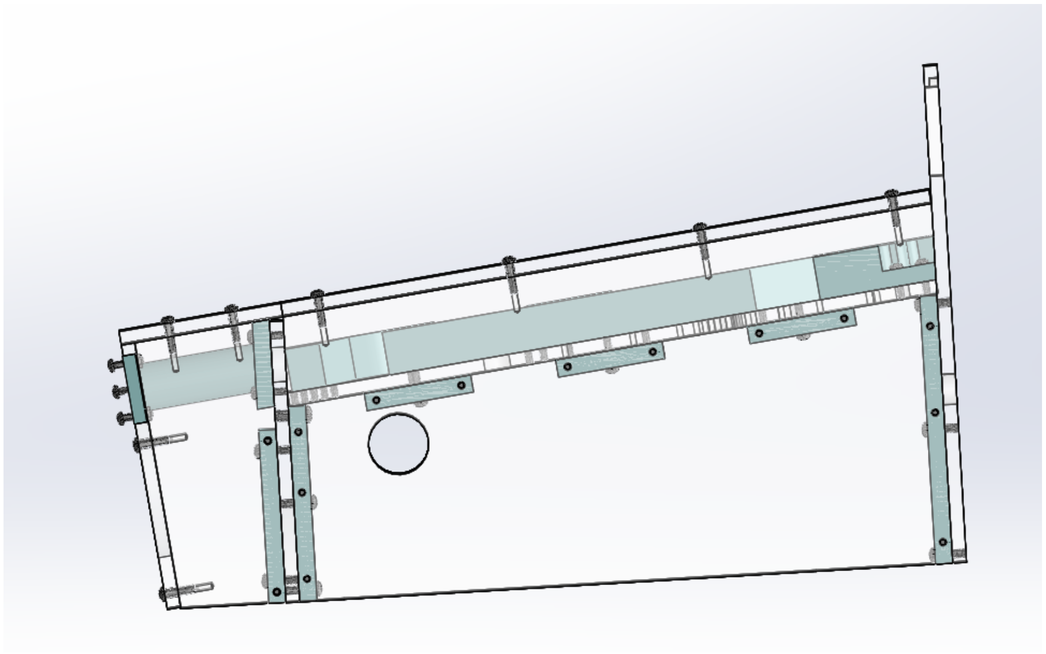
The SOLIDWORKS design of the chassis can be found below in Figure 3. In the figure, it can be observed that the chassis has appropriate openings for mounting the subsystems, as well as an inclined design, and 3D-printed parts (blue) to help with mounting the acrylic sheets (transparent) together. Moreover, the back acrylic sheet and the second sheet from the front have additional openings for the wiring and troubleshooting aid. The original design presented below has an optional top plane made of a transparent acrylic sheet. However, due to our ball not being propelled to the extent that it can escape the playing field, the top plane was deemed excessive. Eliminating it also allowed to decrease production and supply costs.

The height of the border around the playing field is 35 mm, which is ideal for a regular pinball, the diameter of which rarely exceeds 25 mm. The front "box" of the chassis is created to make sure that the plunger will fit well in length and it would offset the height lost in between the playing field and the side walls to make the pinball machine more harmonious. The playing field has a set of 3D-printed parts that allow for a smooth rolling of the ball. As such, the corners are rounded off with the 3D-printed parts and the launching area is divided from the main playing field with a long 3D-printed part.

Acrylic was chosen due to its durability, transparency, and easy drilling and tapping possibilities, which are often inaccessible with wooden materials. The 3D-printed parts were printed with ABS plastic from Notre Dame Engineering Innovation Hub as they proved to be very versatile, and could be integrated with the acrylic material by threading the opening within the 3D-printed part for the appropriate screw size, and then screwed into tapped drilled holes.

As the body of the machine, this subsystem serves as both the physical location and the physical intermediary between the other subsystems, whereas the microcontroller serves as the electrical intermediary. The underside chassis houses the intersection of all electrical connections in the design.





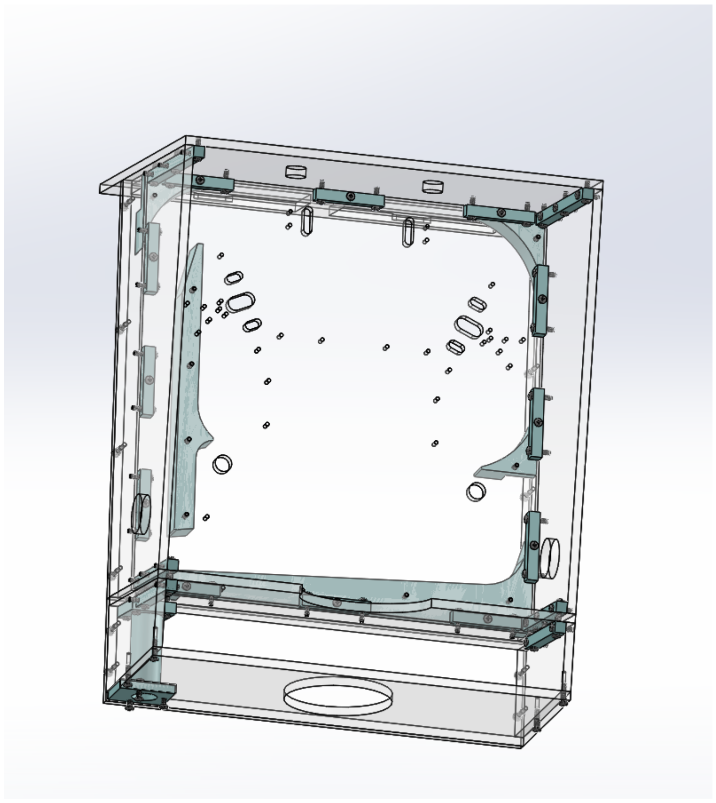


Figure 3. Chassis (diagonal, side, bottom views)

3.4 Detailed Operation of Flipper Subsystem

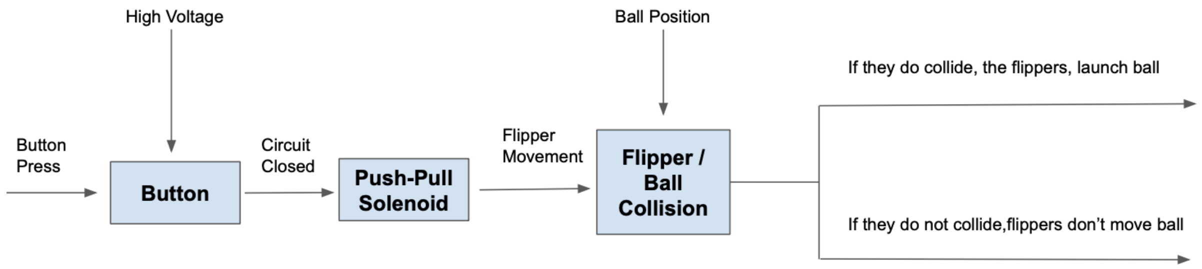
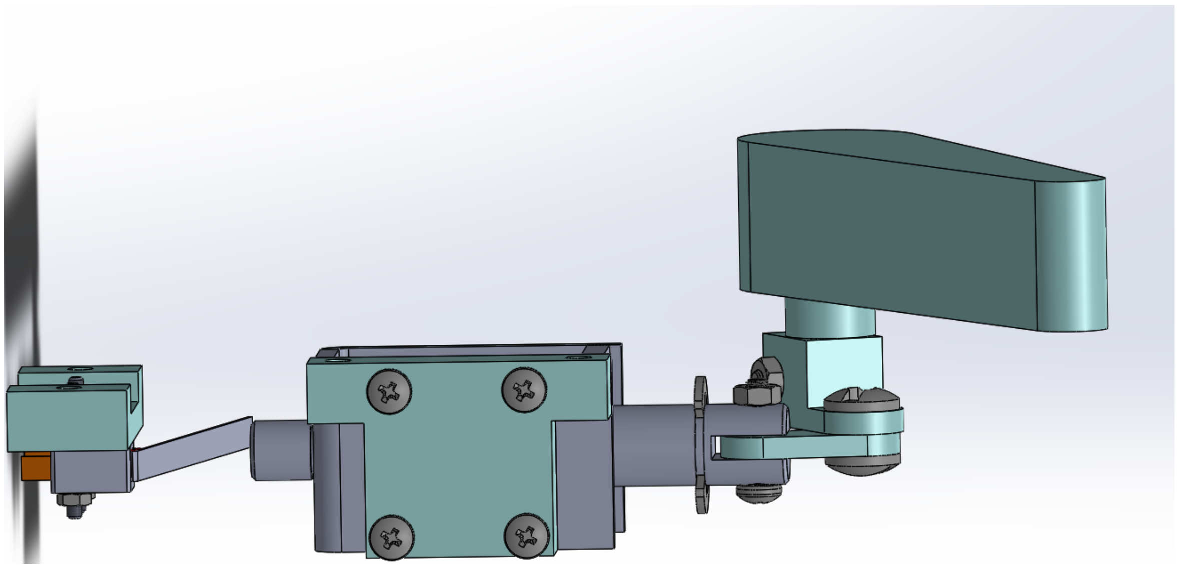


Figure 4. Flipper Subsystem Block Diagram.

The flipper subsystem is a combination of mechanical and electrical components. The purpose of this subsystem is to activate a flipper rotation on the playing field by pressing a button on the side of the pinball chassis. The flipper itself was 3D-printed to accommodate the sizing and design requirements of our pinball machine. We also 3D-printed the supporting and mounting parts for the flipper subsystem in such a way that it would be possible to tighten the flipper at a certain start angle on the playing field and choose how much space we would like to allow for the ball to pass. This part of the flipper subsystem would go directly below the flipper head and below the playing field surface, tightened by a 3M screw and a nut. A small lever system is also present to accomodate for the solenoid compressing movement when activated, this way when the solenoid is activated and the top part of it goes down, the flipper flips up. A more detailed look is presented below in Figure 5. The solenoid is connected to the 3D-printed lever system by a thin metal rod of the M2 diameter, which is used in other subsystems as well.



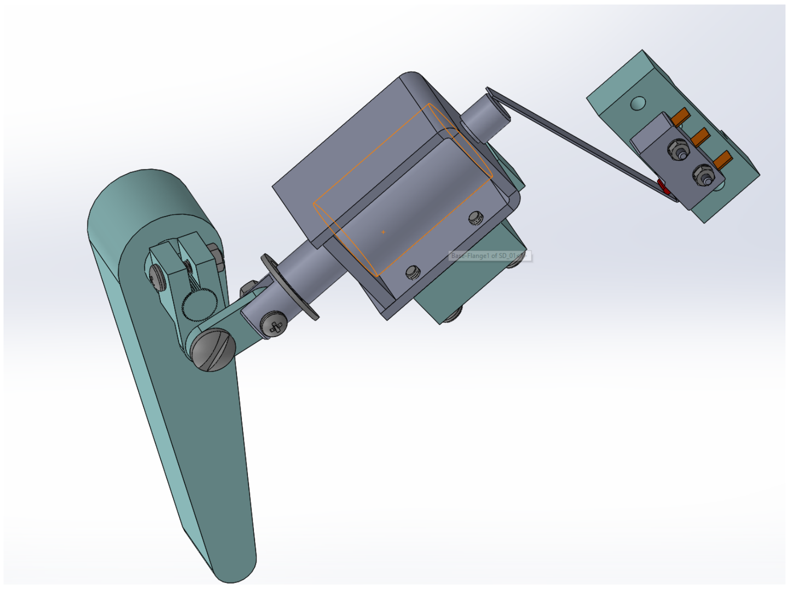


Figure 5. Flipper Subsystem

The solenoid model used for the flippers is Adafruit's "Large Push-Pull Solenoid" (Product ID: 413). The solenoids are attached to pushbuttons, which complete the circuit by connecting the solenoid to the 12 V supply when pressed, which energizes the device and retracts the plunger. The starting force of this model is 6 Newton, which is enough for our flipper's operation, however, a more powerful solenoid could be used in the future to increase compatibility with heavier pinballs. Due to the simplicity of its purpose, our solenoid circuitry is independent of the microcontroller, and the only subsystem it interacts with is the chassis. To prevent large back voltage spikes, an 1N4004 diode was wired in parallel to each solenoid.

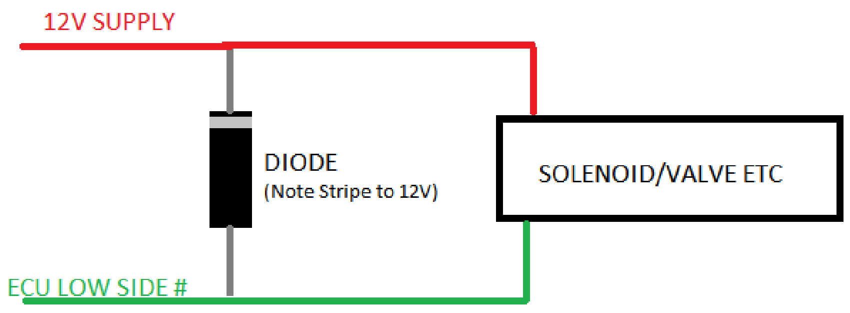


Figure 6. Solenoid Circuit

Figure 6, above, shows our solenoid circuit. Figure 7, below, is the pushbutton model used in our design. Figures 8 and 9 show the complete subsystem integrated into the chassis.

Figure 7. Flipper Push Button Example

The flipper system was relatively simple to test; once they were mounted onto the chassis and wired to the 12 V supply, we could assess functionality by pressing the buttons and observing the flippers, and further test them by seeing how well they propelled the ball.

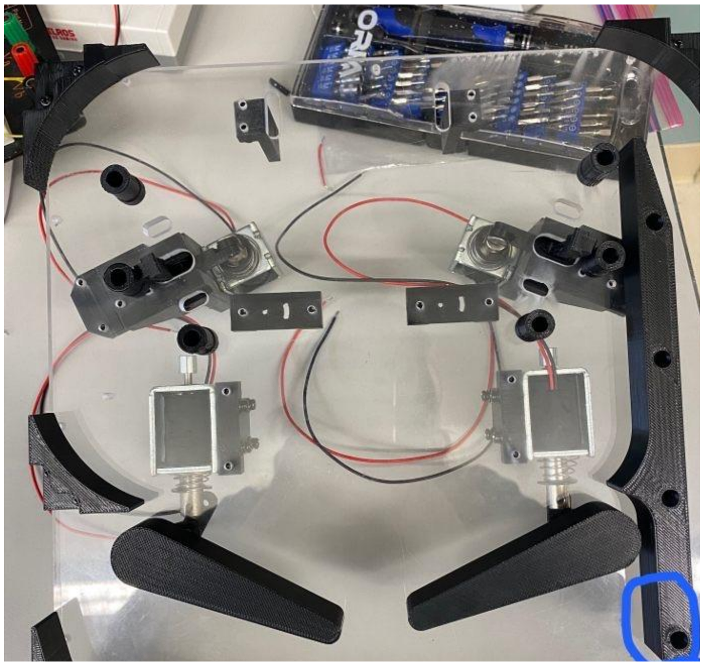


Figure 8. Flipper Subsystem with Less Wires.

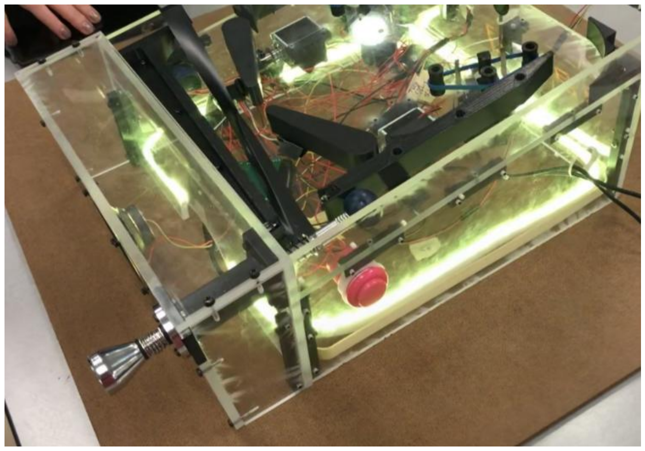


Figure 9. Flipper Subsystem with Wires (Right Side).

3.5 Detailed Operation of Slingshot Subsystem

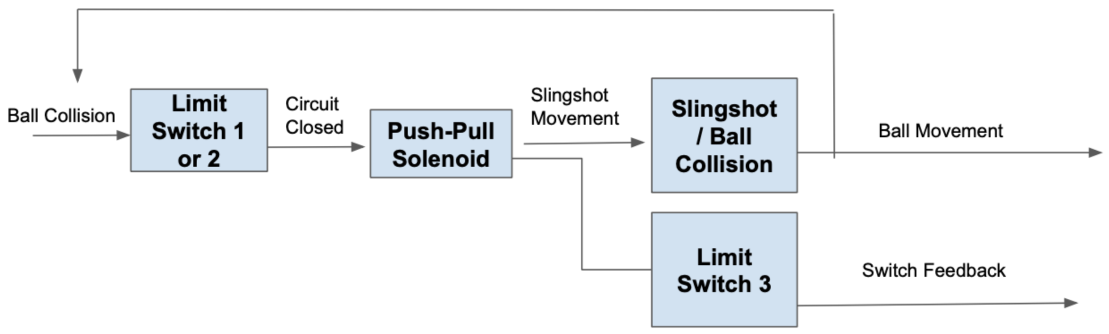


Figure 10. Slingshot Subsystem Block Diagram.

The slingshot subsystem is an integrated electromechanical system. Its purpose is to deflect the ball's path by applying a physical push via a solenoid, which is activated by a limit switch being compressed by the ball. The slingshot is composed of two limit switches, a solenoid, 3D-printed mounting and playing field parts, and a rubber band around the limit switches and the playing field 3D parts. We used two MUZHI SPDT 1NO 1NC 5.5 cm Hinge Lever Momentary Push Button Micro Limit Switches and the same model of solenoid as the flipper for each of the two slingshots. The two switches are wired in parallel to allow for the solenoid to be energized when either of the switches is depressed. The limit switches are wired in the following fashion: the NO pin is connected to the 12 V power supply, the COM pin to ground, and the NC pin is left floating. As before, 12 V is necessary to power the solenoid, and a protection diode is wired to block reverse voltage spikes.

The limit switches and the solenoid are mounted on the 3D-printed mounting system, and the top end of the solenoid is connected to a mechanical 3D-printed system of levers, connected by shoulder screws. A more detailed diagram is presented below in Figure 11.

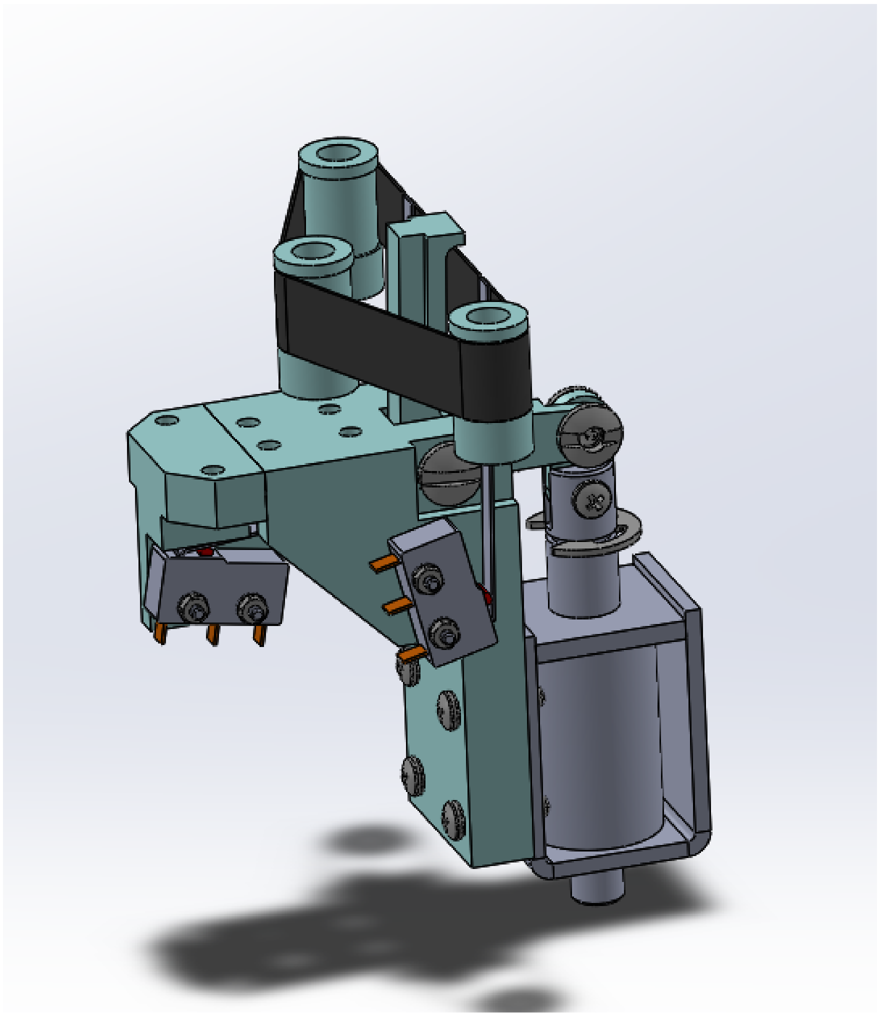


Figure 11. Slingshot Mechanical Connections

The main operation of a slingshot is as follows: when the ball hits the rubber band, it depresses one of the limit switches, which completes the solenoid circuit, energizing it, activating a mechanical relay system, which pushes back against the ball. Two longer rods were used to mount the limit switches parallel to each other on the mounting block, and four nuts in total were used to tighten the system together. The lever system has an angled 3D-printed "kicker" side that connects to the solenoid through a small 3D-printed connection. The "kicker" goes above the playing field to sufficiently propel the ball when the solenoid is activated. The "kicker" also had an axis shoulder screw connection to the 3D-printed mounting part, to ensure it stays in place. The shoulder screws ensure that the movement of the solenoid and the "kicker" are not obstructed by the rough surface while staying at the appropriate axis levels. The solenoid is mounted through predesigned holes on the 3D-printed mounting block using four M3 screws for each solenoid. In our design, the two limit switches are located on two sides of the limit switch, as this design allows for the greatest versatility for getting the ball to compress either of the limit switches, and the lever system to be placed effectively on the solenoid interacting with the playing field.

In the preliminary design, we allocated an additional limit switch at the bottom of the solenoid such that when the solenoid activates, the lower end of the solenoid presses the limit switch. The switch would be connected to the board and trigger an interrupt when the slingshot was activated. This feature was conceived as an additional scoring system, however, due to the inability of the ball to trigger the slingshot, we have discarded this feature in this prototype.

We considered the slingshot subsystem working based on the successful activation of slingshots when either limit switch is depressed, which needed to be verified for both switches. To assist with wiring the limit switches correctly, we used a multimeter to verify connectivity before connecting the devices to the power rail. Despite the ball's inability to press the switches, due to each slingshot having a rubber band around it to help increase the area the ball can hit the switch from, the ball still bounces away from the slingshot well due to the rubber band's elasticity, making these devices useful obstacles in the game. The complete slingshot subsystem is shown below in Figure 12.

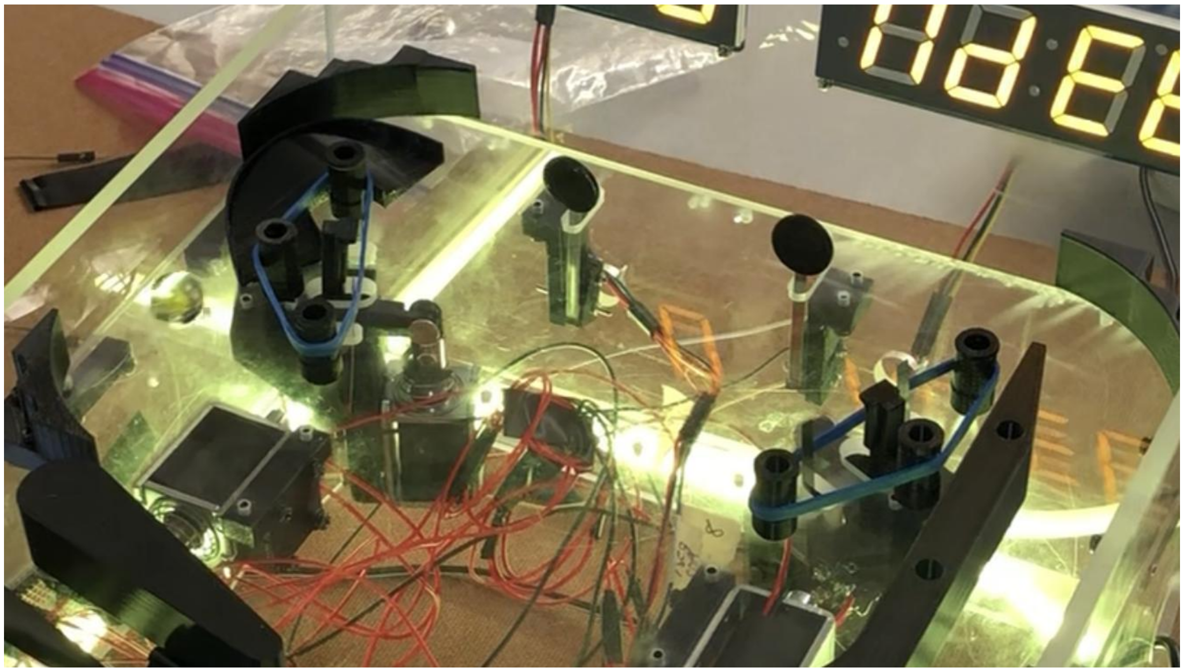


Figure 12. Slingshot Subsystem.

3.6 Detailed Operation of Ball Launcher Subsystem

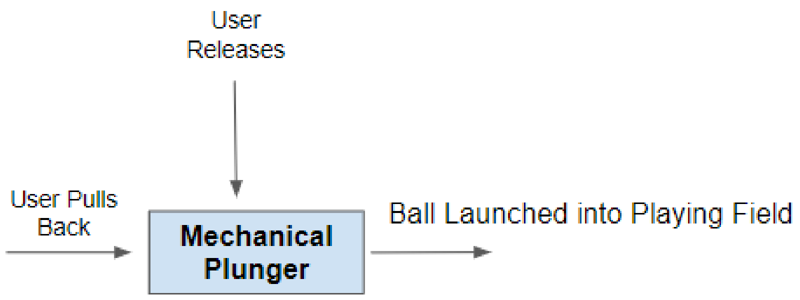


Figure 13. Ball Launcher Subsystem Block Diagram.

The ball launcher subsystem is a purely mechanical design that consists of the plunger and the gutter that it sits in. Its purpose is to launch the ball into the playing field to begin a new game after it has rolled back from the end of the previous one. This subsystem was required to be durable and robust, and allow the player to control the strength of their launch, which affects the speed of the ball, an important part of the strategy of the game. For this reason we chose a purely mechanical solution, as it affords the fine control a user will need while being durable, as the steel plunger and its spring can handle repeated wear and tear. Due to its mechanical nature, the only subsystem it interacts with is the chassis, which it is screwed into through a hole cut into the acrylic to accommodate the plunger’s size.

The operation of the ball launcher is as follows: the user pulls back the plunger, compressing the spring and allowing the ball to pass through into the gutter and rest on top of the spring. The user determines how much force they wish to impart with their launch, and pulls the plunger back accordingly. They then release the plunger, decompressing the spring and pushing the ball against gravity, up through the gutter and around the curved corner into the playing field. This subsystem was tested once the chassis was fully assembled. We placed a ball on the spring, and tried launching it with various amounts of force. Once the ball made it successfully into the playing field, we considered the subsystem successfully operational.

Our design fulfills the requirements of this subsystem, with the only unresolved issue coming from the fact that the ball does not always roll into the gutter when the plunger is pulled back; often the user must manually place the ball on the plunger instead. This issue comes from the design of the chassis itself, and the lack of a diagonal incline for the ball to roll into. A solution to this error would be to design an incline into the bottom of the playing field that permits the ball to roll fully into the gutter without user intervention.

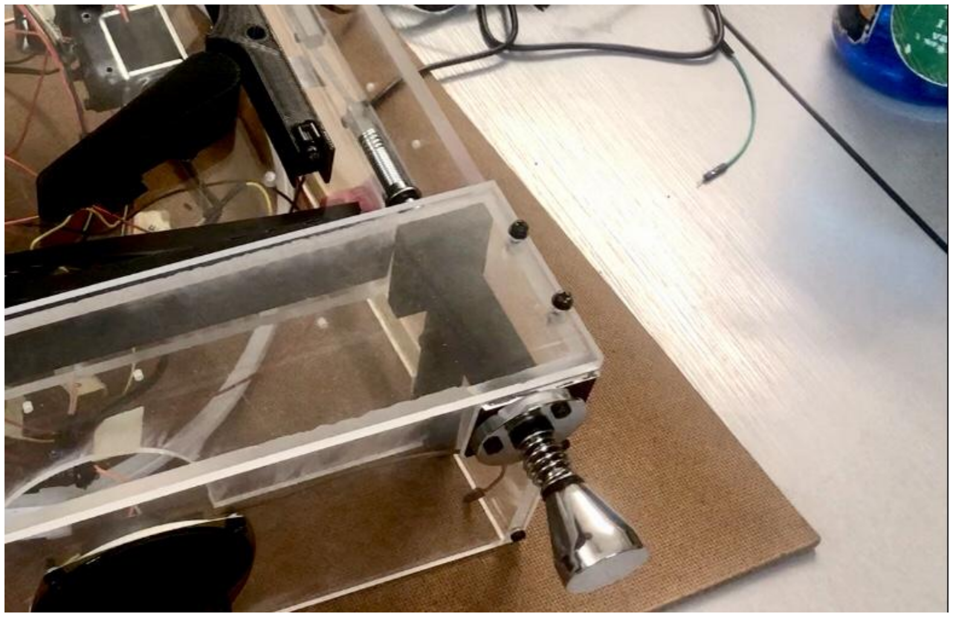


Figure 14. Ball Launcher Subsystem.

3.7 Detailed Operation of Target Subsystem

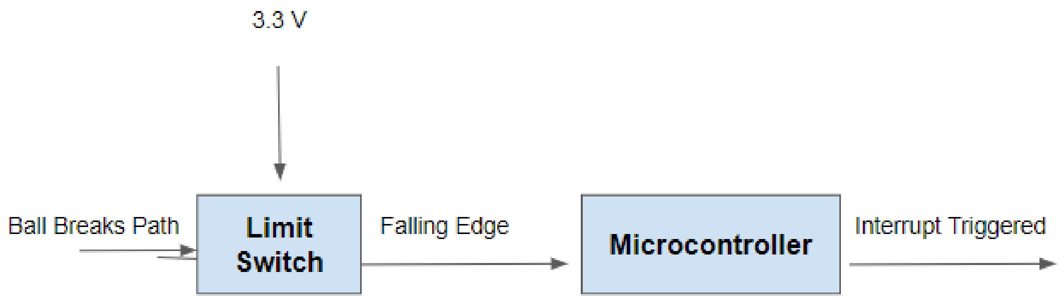


Figure 15. Target Subsystem Block Diagram.

The target subsystem is a combination of a mechanical and electrical design. The primary electromechanical component of the subsystem is a limit switch. A small round plate is attached to the end of the metal compressor of the limit switch and then the entire system is mounted underneath the playing ground leaving the mounted plate in the playing field to be hit by the ball. We 3D-printed a small, (2 cm diameter) round plate to be attached to the end of the metal part of the limit switch, and the same process was used to create the mounting. The common pin of the limite switch was connected to a microcontroller I/O port equipped for falling edge interrupts, with the NC (normally closed) pin connected to 3.3 V, and the NO (normally open) pin connected to ground. Normally, the microcontroller sees 3.3 V and does not trigger an interrupt. However, when the switch is closed, it changes the state visible on the common pin to 0, which presents a falling edge, triggering the interrupt. Specifically, we chose the MUZHI SPDT 1NO 1NC 5.5cm Hinge Lever Momentary Push Button Micro Limit Switch for our project. Figure 16, below, shows the pin layout of this device.

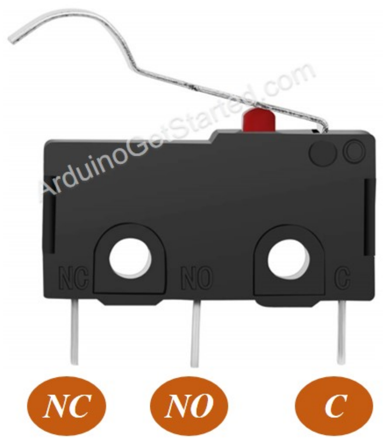


Figure 16. Limit Switch Pin Layout.

The switch is mounted on the 3D-printed mounting system at a slight angle so that a small press of the ball is able to compress the switch and activate the target response. In order to do so, the 3D-printed system required an elliptical slot in it for mounting the limit switch and some testing and adjustments for the switch placements. The switch was mounted using two 2M rods, fixed on the mounting system by a nut on each side.

If the ball was able to hit the target and compress the limit switch, we considered the system fully operational. At the moment, two switches correspond each to a single pin, however, the system could be simplified by wiring both switches in parallel and sending the ground signal to a singular pin on the microcontroller designated for scoring interrupts.

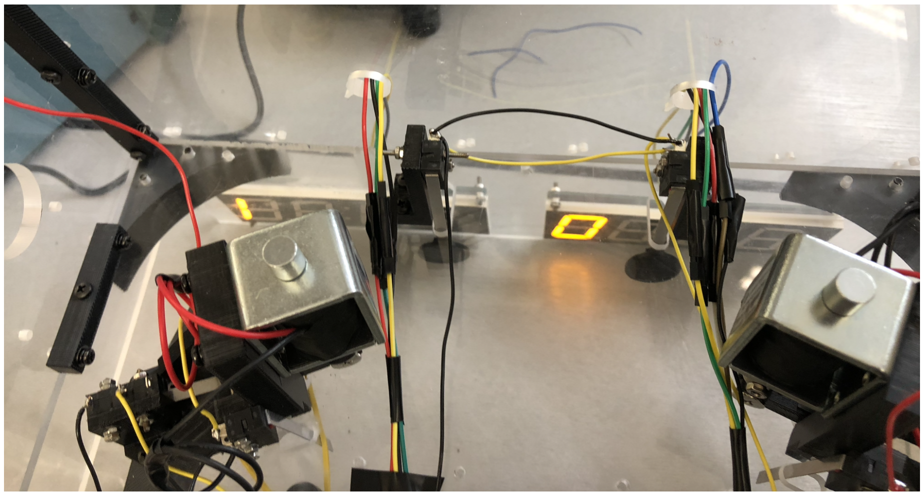


Figure 17. Targets Subsystem (Bottom View).

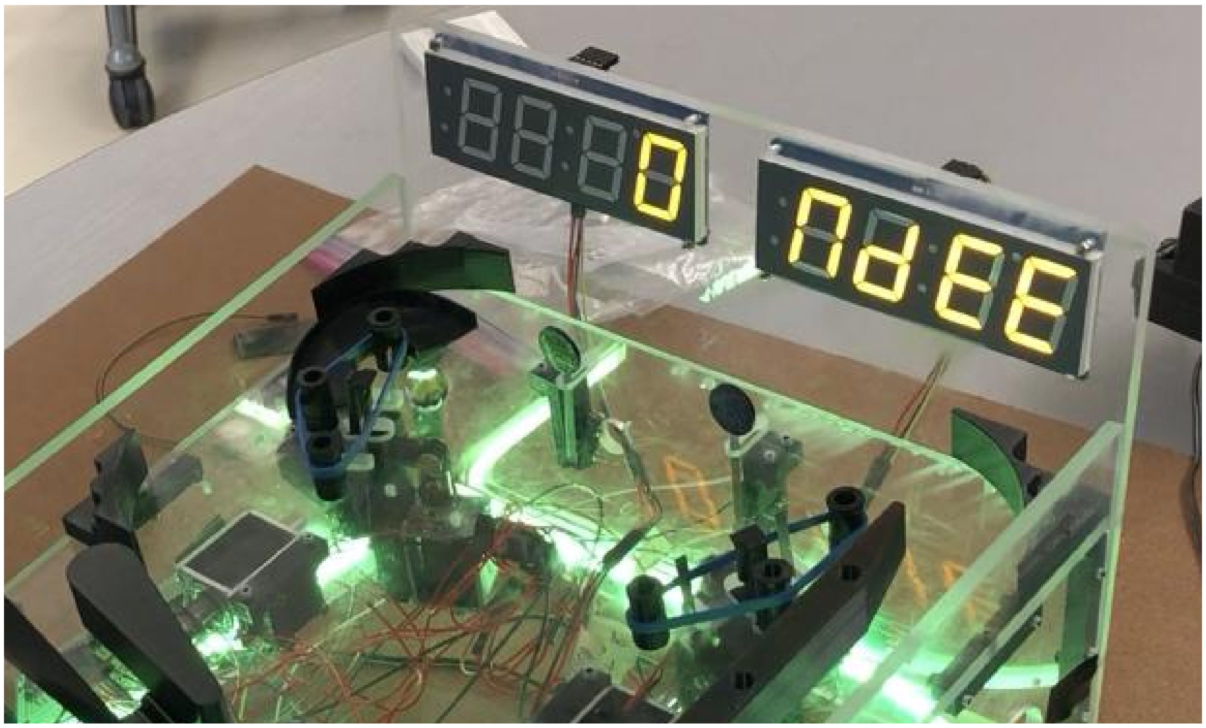


Figure 18. Targets Subsystem (Top View).

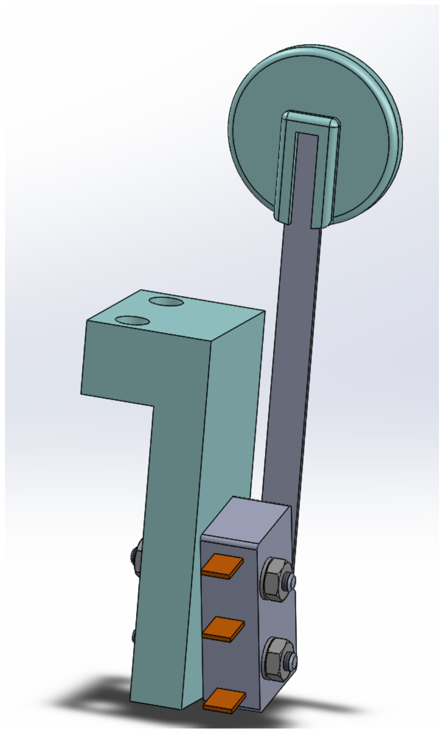


Figure 19. Target Subsystem (SolidWorks).

3.8 *Detailed Operation of Beam-Break Subsystem*

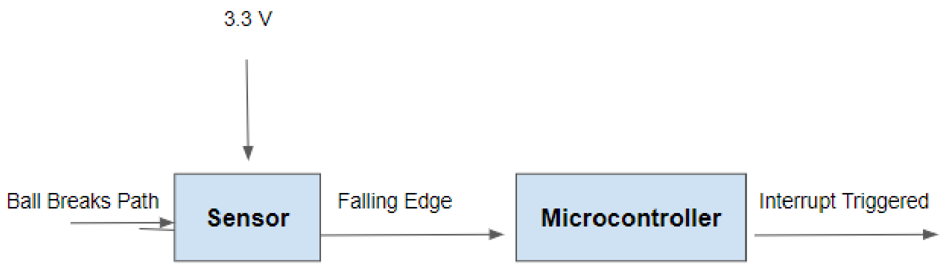


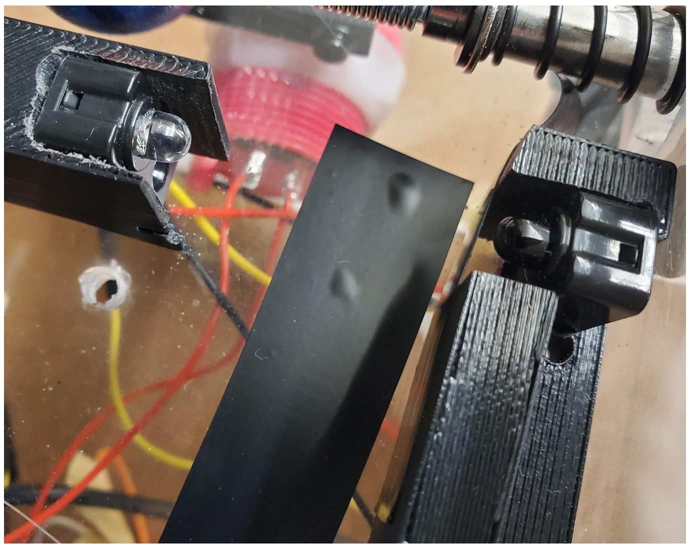
Figure 20. Beam-Break Subsystem Block Diagram.

The Beam-Break is primarily an electrical subsystem, with a simple one port connection to the microcontroller. The emitter continuously sends an infrared (IR) signal to the receiver, which is sensitive to that same light. When something passes between the two, and is not transparent to IR, then the 'beam is broken' and the receiver will send a low signal through its data line, rather than the high (3.3 V) signal that is normally present. This falling edge can then trigger an interrupt on the microcontroller, leading to the game-over routines.

Both the emitter and receiver are connected to ground and the 3.3 V supply line. The receiver also has a data line, connected to an I/O port on the microcontroller configured for falling edge interrupts. Additionally, as the collector of the output transistor needs to have a pull-up resistor present, we enabled the internal pull-up resistor of the microcontroller pin used by the data line.

In our design, the emitter is placed inside a structural 3D-printed part that divides the launch path from the playing field. The receiver is placed inside an inclined 3D-printed structural part on the bottom of the playing field in such a way that when the ball is approaching the plunger area, it passes through the beam, but otherwise, the emitter and receiver see one another, and the beam is intact.

We considered the beam-breaker subsystem functional when the ball successfully passed through the beam and activated the attached interrupt, triggering audio and a different LED pattern. The mounted gutter sensor can be seen below in Figure 21.



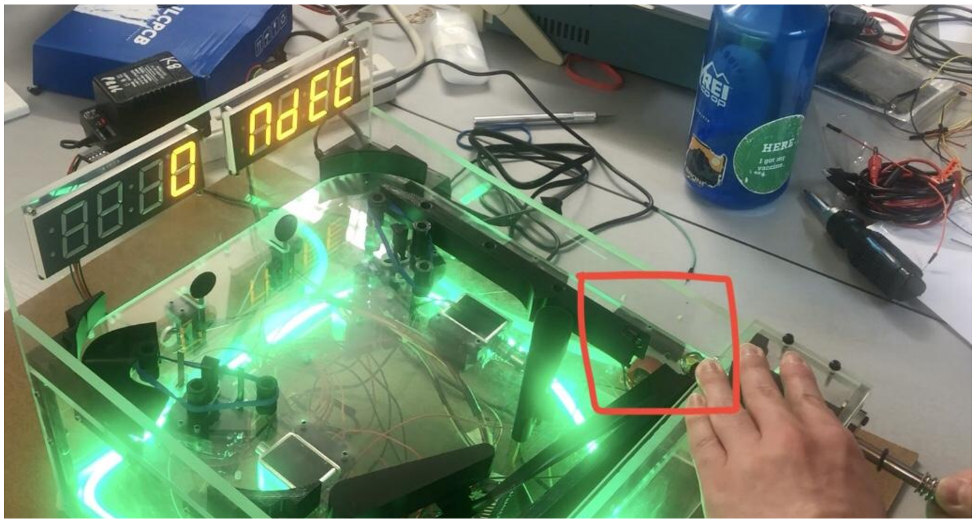


Figure 21. Gutter Sensor (Left View).

3.9 Detailed Operation of LED Subsystem

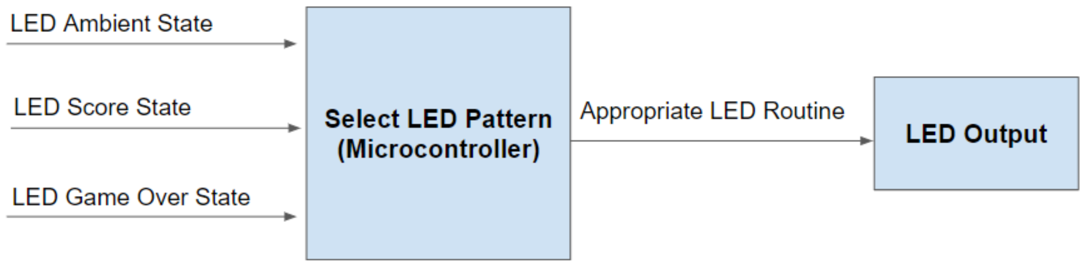


Figure 22. LED Subsystem Block Diagram.

The LED subsystem is primarily an electrical subsystem, with a simple software component governing its operation. Its purpose is to provide ambience during play, and a visual reaction when events occur; specifically, these events are scoring and the end of the game. The only requirements for this subsystem were that it be controllable by the microcontroller's 3.3 V logic levels in some way, and that it be bright enough to stand out from within the machine's chassis. To accomplish this, we chose to use an iPixel RGB flexible LED strip, which was run off of the 12 V supply. As it was run off of the 12 V supply, IRLB8721PbF power NMOS transistors were used to allow a 3.3 V control signal from the microcontroller to switch between 12 V and ground, as well as insulate from the hundreds of microamps drawn by the LED strip. We chose to use a 12 V strip because it possessed enough internal LED's to have the brightness needed to illuminate the entire chassis, and we chose a four-terminal RGB strip because it allowed us to manipulate the color of the strip solely through altering the voltage present at each control pin, which can be accomplished via software. We chose to use these particular power MOSFETs because their gate threshold voltage (~1.8 V) was optimal for 3.3 V microcontroller logic, they can handle a V_{DS} within our 12 V range (30 V maximum) with low resistance (tens of mΩ), and draw very low gate leakage current (~100 nA). With these criteria, our MOSFETs can be controlled by our microcontroller without threatening its pins by excess current draw, while not adding significant resistance between the LED and voltage supply that could reduce the power available to the strip. The relevant part of the schematic concerning the LED subsystem can be found below in Figure 23.

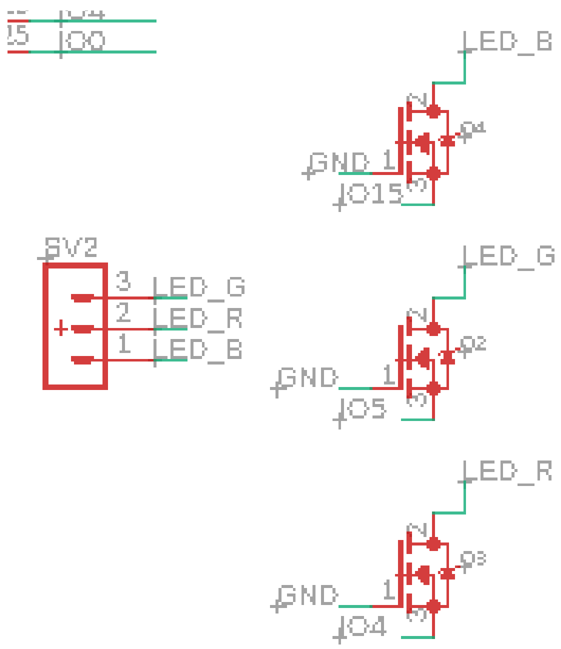
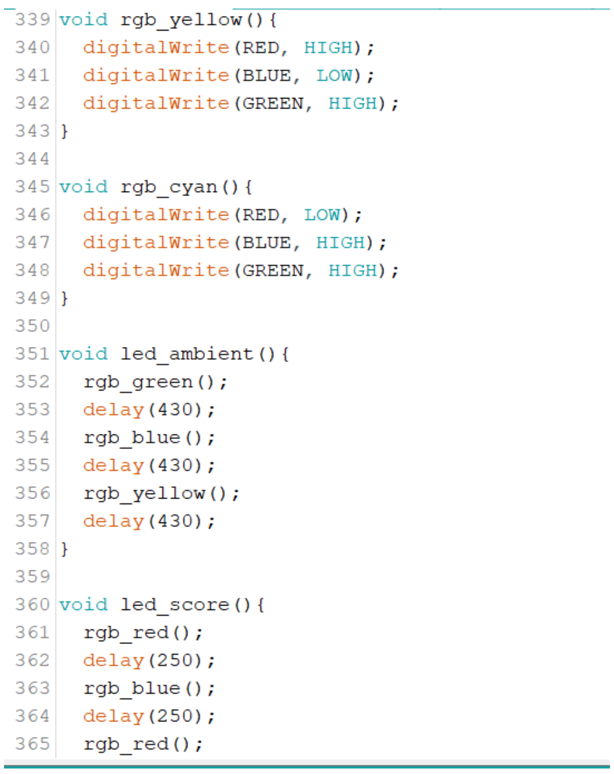


Figure 23. LED Schematic.

The LED strip is a common-anode model, with the anode connected to the 12 V power rail, and the R, G, and B ports connected to the drain of their respective NFET's. Three I/O ports, labeled the R, G, and B I/O ports on our schematic, are connected to their respective NFET's gate, and each source is connected to ground. When a port on the LED strip is connected to ground, that color is activated, which we achieve by sending a high signal to the NFET's gate, which closes the electrical switch and connects the pin to ground. Otherwise, when a low signal is present at the gate, the switch is open, and the floating port prevents that color from turning on. There are a total of eight permutations of ports as on or off, corresponding to seven colors and the strip being turned off. For example, turning on the red and blue ports while leaving the green port off creates a magenta output, as the two wavelengths mix together and are interpreted as a third color by the human eye due to constructive interference.

Within the software, these signals are sent as digital outputs from I/O ports, using the `digitalWrite` function. HIGH signals correspond to 3.3 V and the closing of a switch leading to the activation of that color, and LOW signals correspond to 0 V and the opposite behavior. To simplify the main program, custom functions were written to store each individual color (as well as turning the LED off), and further functions created to reference these color functions, with delays separating them to make the LED flash. This way, within the main function or LED task, an entire routine of the LED corresponding to the current state of the game can be enacted with a single function call. A sample of these color functions and routine functions can be found below in Figure 24.



```
void rgb_yellow(){
  digitalWrite(RED, HIGH);
  digitalWrite(BLUE, LOW);
  digitalWrite(GREEN, HIGH);
}

void rgb_cyan(){
  digitalWrite(RED, LOW);
  digitalWrite(BLUE, HIGH);
  digitalWrite(GREEN, HIGH);
}

void led_ambient(){
  rgb_green();
  delay(430);
  rgb_blue();
  delay(430);
  rgb_yellow();
  delay(430);
}

void led_score(){
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_red();
```

Figure 24. LED Code Sample.

The LED subsystem was originally tested using the ESP-32 DevKit, where the color functions were written into a loop, and the output of three I/O ports went to the gates of the same transistors, with the 12 V supply provided by a DC lab power supply. When the PCB board was built, the LED strip was wired to the RGB ports and the 12 V rail, and we observed to see if the LED glowed the correct colors when the power was turned on (ambience). When ambience was confirmed, we knew the LED subsystem was functioning as intended. This subsystem fulfills all of its requirements, and does not require further improvements in a future iteration.

3.10 Detailed Operation of Audio Subsystem

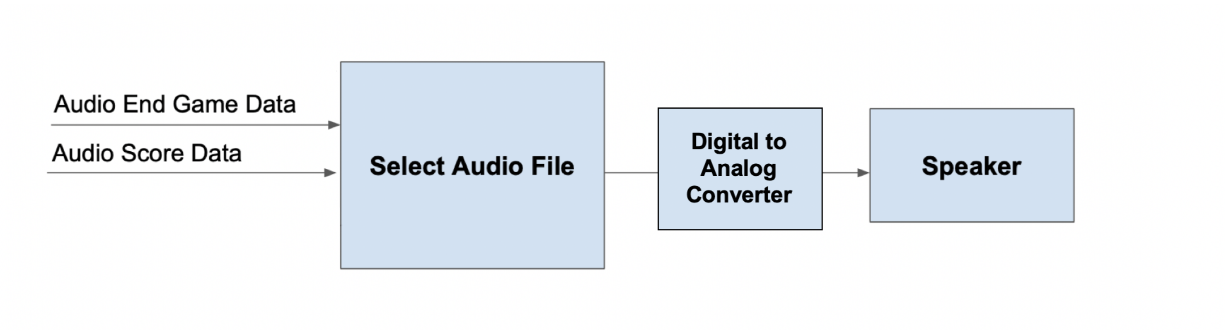


Figure 25. Audio Subsystem Block Diagram.

The audio subsystem consists of several parts. Its primary function is to elevate the player's game-playing experience by producing a sound in response to any events that result in a score change, such as the pinball hitting a target or game over. It was critical to ensure that the selected speaker, amplifier, and DAC would be compatible with the microcontroller and its supported communication protocols. The microcontroller we used contains two Inter-IC Sound (I2S) peripherals. I2S is a serial, synchronous communication protocol that allows for the transmission of audio data between two digital devices. We selected an I2S audio breakout board that uses the MAX98357A digital to analog converter. This DAC converts I2S audio to an analog speaker to drive the chosen speaker, which has a 4 Ohm impedance and a maximum power output of 3 W. This speaker has a diameter of 3", making it a suitable size for the physical build constraint. The MAX98357A has a built-in class D amplifier which can provide 3.2 W output power into a 4 Ohm load at 5 V, pairing well with the chosen speaker. Several other benefits of the MAX98357A are short circuit and thermal protection, having 92% efficiency, not requiring a MCLK, and reducing click and pop-up noise.

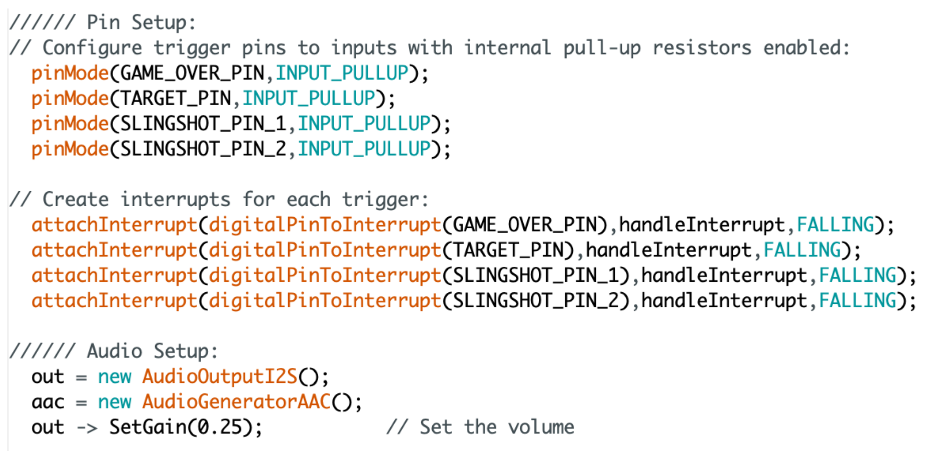
The I2S audio breakout board only requires a few pins: LRCLK (left/right clock input), BCLK (bit clock input), DIN (serial data input), GAIN (gain setting), SD (shut down and channels elect), GND, and VDD, which had a supply voltage ranging from 2.5 V to 5.5 V, making it ideal for our 3.3 V logic level. The breakout board was appropriately connected to the positive and negative outputs of the speaker, remaining in its default "mono" operation, since we only drive a single speaker. Further, the gain of the amplifier can be configured in several ways if we desired. For our design this pin was left unconnected, thus defaulting to +9 dB, which through testing we determined was an appropriate volume.

The ESP8266 Audio Arduino Library allowed for a variety of audio formats to be played and was installed from GitHub. We wanted our pinball machine to mimic an arcade game, so we downloaded several sound effects online as a WAV file to play when different events in the game occurred. To prepare the audio files to be compatible with the audio hardware and software, we first converted each sound effect to be an 8-bit, 8 Hz, Mono WAV file. Then, we converted each WAV file to an AAC file. This Advanced Audio Coding format was chosen for its characteristics of being lossy, compressed, and efficient while balancing the audio quality and portability. Finally, the AAC files were converted to hex and formatted as header files. The files were stored in the Arduino's program memory (`PROGMEM`), a non-volatile flash memory where the uploaded sketch resides. The I2S output was initialized using the line:

```
out = new AudioOutputI2S();
```

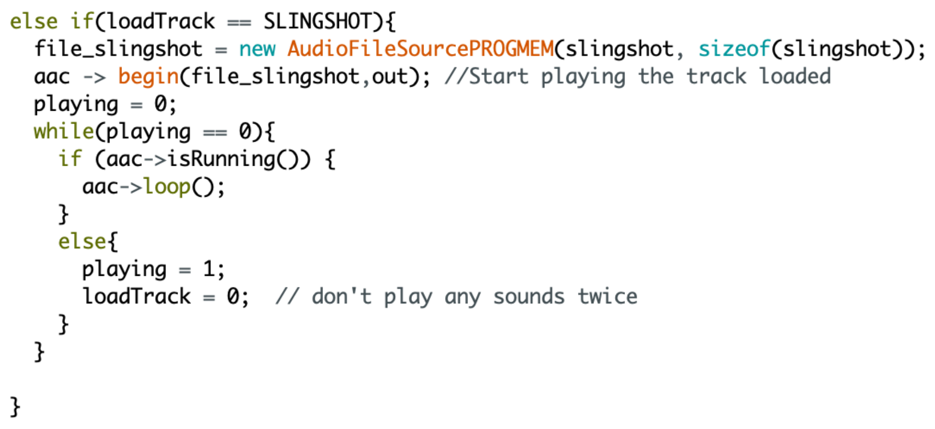
After testing several values, the appropriate software gain of the output was chosen as 0.25.

A different sound effect was played in response to the triggering of the slingshots, targets, and beam break gutter sensor. When an interrupt on an appropriate pin is triggered, a state variable called `loadTrack` is set, which can then be read by the audio task. When the audio task reads this value, it loads the appropriate track in from its header file, and begins playing the file. A sample of the audio subsystem code is shown below in Figures 26 and 27.



```
////// Pin Setup:
// Configure trigger pins to inputs with internal pull-up resistors enabled:
  pinMode(GAME_OVER_PIN,INPUT_PULLUP);
  pinMode(TARGET_PIN,INPUT_PULLUP);
  pinMode(SLINGSHOT_PIN_1,INPUT_PULLUP);
  pinMode(SLINGSHOT_PIN_2,INPUT_PULLUP);

// Create interrupts for each trigger:
  attachInterrupt(digitalPinToInterrupt(GAME_OVER_PIN),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(TARGET_PIN),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(SLINGSHOT_PIN_1),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(SLINGSHOT_PIN_2),handleInterrupt,FALLING);

////// Audio Setup:
  out = new AudioOutputI2S();
  aac = new AudioGeneratorAAC();
  out -> SetGain(0.25);           // Set the volume
```

Figure 26. Audio Setup Code Sample.



```
else if(loadTrack == SLINGSHOT){
  file_slingshot = new AudioFileSourcePROGMEM(slingshot, sizeof(slingshot));
  aac -> begin(file_slingshot,out); //Start playing the track loaded
  playing = 0;
  while(playing == 0){
    if (aac->isRunning()) {
      aac->loop();
    }
    else{
      playing = 1;
      loadTrack = 0;  // don't play any sounds twice
    }
  }

}
```

Figure 27. Audio Event Trigger Code Sample.

The audio subsystem was initially tested using the ESP32-DevKit with limit switches to trigger interrupts. It was evident when the subsystem was functioning correctly when pressing the pushbutton played the corresponding audio track without clicking on running on without stopping. The audio subsystem fulfills all of the outlined requirements, and its software can be easily modified to accommodate new sounds just by adding the appropriate header files. Additionally, ambient noise is supported through use of the audio task, but was removed due to poor feedback from users regarding its monotony.

3.11 Detailed Operation of 7-Segment Display Subsystem

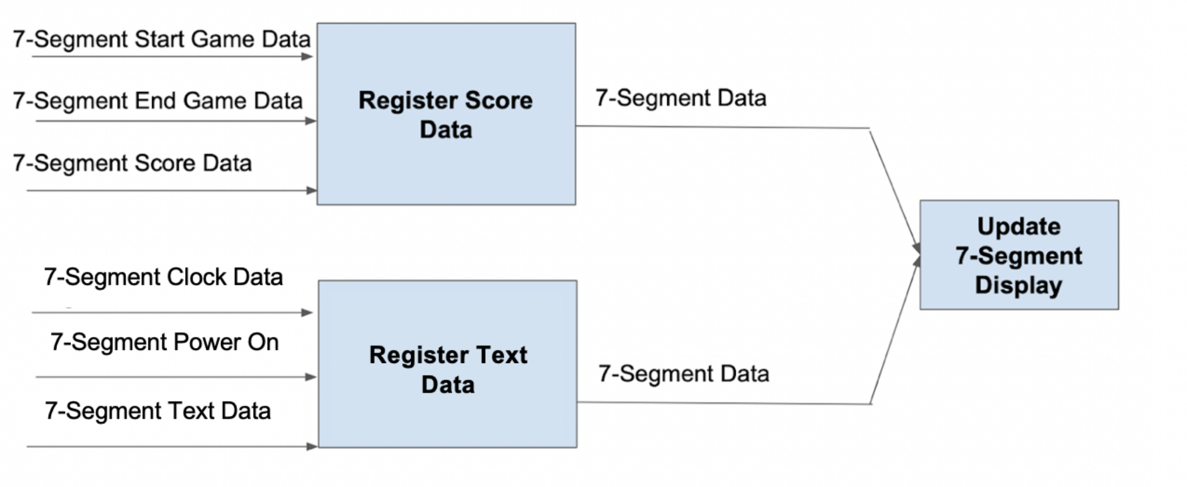


Figure 28. 7-Segment Display Subsystem Block Diagram

The 7-Segment Display served two purposes: display and update the game score as the user interacted with our scoring subsystemson on the first board, and print a decorative message on the second. In order to transmit these pieces of information to the boards, the subsystem used an Inter-Integrated Circuit (I2C) communication protocol to send data from the microcontroller to the 7-Segment Display. This communication protocol is different from the one used for the audio subsystem (I2S), so some troubleshooting needed to be done in the form of organizing the use of particular pins and the manner in which code was to be processed.

The 7-segment display sits in a backpack that uses five pins: V_{logic} (logic level - set to 5 V), VDD (5 Volt supply), GND (ground), SCL (serial clock line), SDA (serial data line). Both the 5 V and VDD pins were connected to a 5 V power supply, despite the microcontroller operating at a logic level of 3.3 V (it did not work when the logic level was connected to 3.3 V). This 4-digit 7-segment display has LED matrices that are ‘multiplexed’ – containing 14 pins for controlling each section of the 4 digits. This model already contains a driver chip with a built-in clock for the multiplexing display as part of the backpack. There are current drivers that are operating continuously for the brightness and color of its LED's, with the additional option of address-selection jumpers for connecting up to eight 7-segments on one I2C bus.

There were multiple libraries that aided in optimal communication between the microcontroller and the displays. These libraries were provided by Adafruit, and include functions for both I2C setup and writing various characters. Due to the way the backpack's are wired, they both possessed the same I2C address. Therefore, it was necessary to use two separate buses to talk to them separately. Using the `Wire` library and the `TwoWire` function, we were able to create separate buses using the GPIO pins of the microcontroller; this setup can be seen below in Figure 30.

The 7-segment display is updated within the main loop whenever the primary state variable indicates a score-changing event, in the same manner that LED routines are triggered. By contrast, the message right-hand message display remains the same throughout, and its message can be changed by modifying one string in the code (for our demonstrative purposes it says "NDEE"). Samples of 7-segment display code are shown below in Figures 29-31.

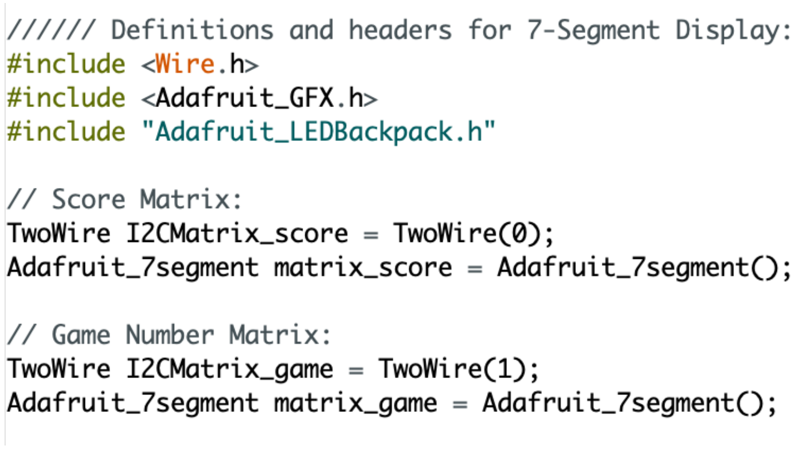
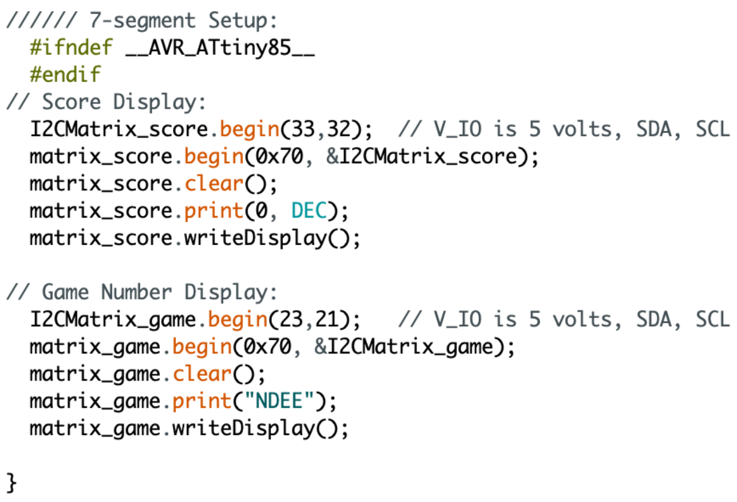
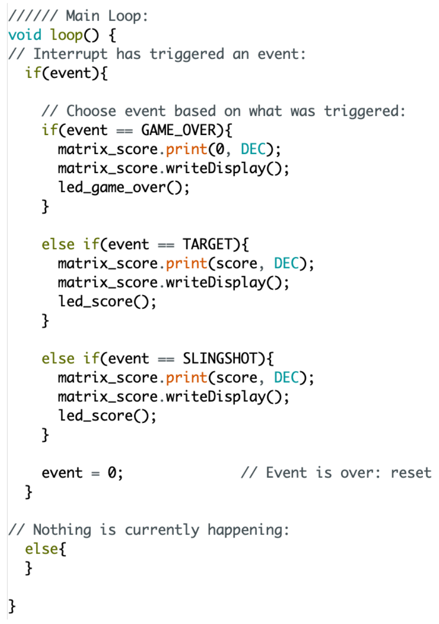

```
////// Definitions and headers for 7-Segment Display:
#include <Wire.h>
#include <Adafruit_GFX.h>
#include "Adafruit_LEDBackpack.h"

// Score Matrix:
TwoWire I2CMatrix_score = TwoWire(0);
Adafruit_7segment matrix_score = Adafruit_7segment();

// Game Number Matrix:
TwoWire I2CMatrix_game = TwoWire(1);
Adafruit_7segment matrix_game = Adafruit_7segment();
```

Figure 29. 7-Segment Display Definitions and Headers Code Sample.



```
////// 7-segment Setup:
  #ifndef __AVR_ATtiny85__
  #endif
// Score Display:
  I2CMatrix_score.begin(33,32);  // V_IO is 5 volts, SDA, SCL
  matrix_score.begin(0x70, &I2CMatrix_score);
  matrix_score.clear();
  matrix_score.print(0, DEC);
  matrix_score.writeDisplay();

// Game Number Display:
  I2CMatrix_game.begin(23,21);   // V_IO is 5 volts, SDA, SCL
  matrix_game.begin(0x70, &I2CMatrix_game);
  matrix_game.clear();
  matrix_game.print("NDEE");
  matrix_game.writeDisplay();

}
```

Figure 30. 7-Segment Display Setup Code Sample.



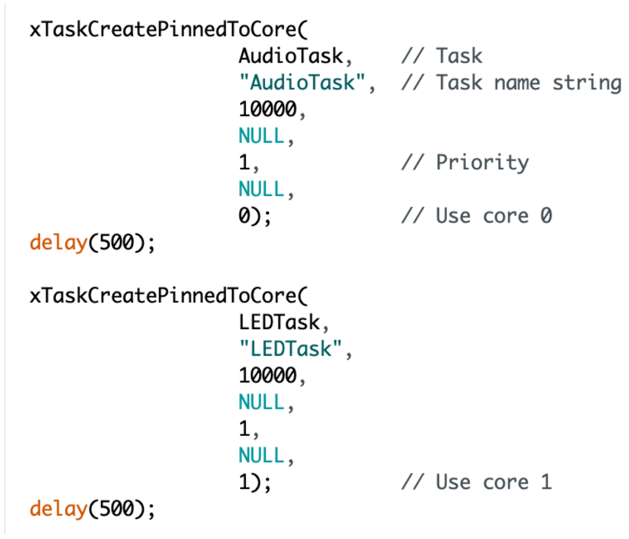
```
////// Main Loop:
void loop() {
// Interrupt has triggered an event:
  if(event){

    // Choose event based on what was triggered:
    if(event == GAME_OVER){
      matrix_score.print(0, DEC);
      matrix_score.writeDisplay();
      led_game_over();
    }

    else if(event == TARGET){
      matrix_score.print(score, DEC);
      matrix_score.writeDisplay();
      led_score();
    }

    else if(event == SLINGSHOT){
      matrix_score.print(score, DEC);
      matrix_score.writeDisplay();
      led_score();
    }

    event = 0;              // Event is over: reset
  }

// Nothing is currently happening:
  else{
  }

}
```

Figure 31. 7-Segment Display Event Trigger Code Sample.

The 7-segment display was initially tested using an Arduino setup and environment due to time constraints for deliverables at the time. Upon further development with testing this subsystem, the libraries designed by Adafruit for their 7-segment display product could be transferred for use and testing on the ESP32-DevKit. There were some issues present with printing and updating the specific values desired for the scope of the project, but this was quickly resolved once the wiring of the systems were properly integrated, and the `Wire` library properly deployed.. The 7-segment display subsystem fulfills all of its requirements and adds additional functions by both keeping and incrementing the game's score, as well as displaying a chosen message. We originally wanted to keep track of the number of runs left before game over, but given the scope of the project, and no start or stop button, the immediacy of the game rendered a game counter irrelevant and a waste of the digits available - we felt a 4-digit message was a better use of the second 7-segment display.

3.12 Interfaces

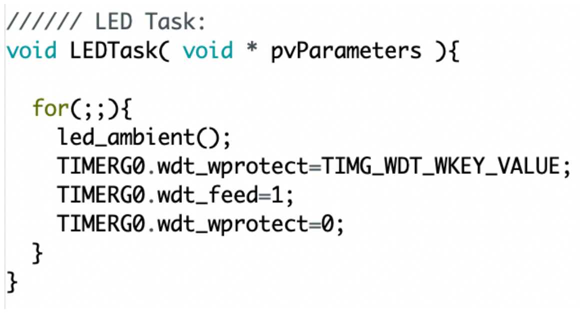
When testing the audio, 7-segment, and LED subsystems together, it was evident that the events were not occurring simultaneously but instead, sequentially. Pinball is a fast-paced game; we wanted the score and lights to change while the system plays an audio track. This multitasking was accomplished using FreeRTOS. Our ESP-32 microcontroller has a dual-core processor, allowing tasks to be separated across cores. FreeRTOS rapidly switches between tasks, giving the impression that the system is multitasking. Two tasks were created and pinned to two separate cores: an audio task and an LED task. Several parameters were specified to create and pin tasks, including allocating the stack depth, setting the task priority, and the index number of the CPU that the task should be pinned to. The two created tasks are in Figure 32 below.



```
xTaskCreatePinnedToCore(
                  AudioTask,    // Task
                  "AudioTask",  // Task name string
                  10000,
                  NULL,
                  1,            // Priority
                  NULL,
                  0);           // Use core 0
delay(500);

xTaskCreatePinnedToCore(
                  LEDTask,
                  "LEDTask",
                  10000,
                  NULL,
                  1,
                  NULL,
                  1);           // Use core 1
delay(500);
```

Figure 32. FreeRTOS Creating and Pinning Tasks to Dual-Core Code Segment.

The Audio task was pinned to core 0. This task consisted of the entire audio instruction for each interrupt. The LED task, which was pinned to core 1, consisted only of the ambient LED sequence. The alternative LED sequences and 7-segment display updates remained in the main loop, where the behavior was determined by checking a state variable. To prevent the ESP-32 from resetting, the watchdog timer was fed as part of the LED task, as shown in Figure 33 below.



```
////// LED Task:
void LEDTask( void * pvParameters ){

  for(;;){
    led_ambient();
    TIMERG0.wdt_wprotect=TIMG_WDT_WKEY_VALUE;
    TIMERG0.wdt_feed=1;
    TIMERG0.wdt_wprotect=0;
  }
}
```

Figure 33. Feed WDT in LED Task Code Segment.

The complete software logic flow is shown below in Figure 34.

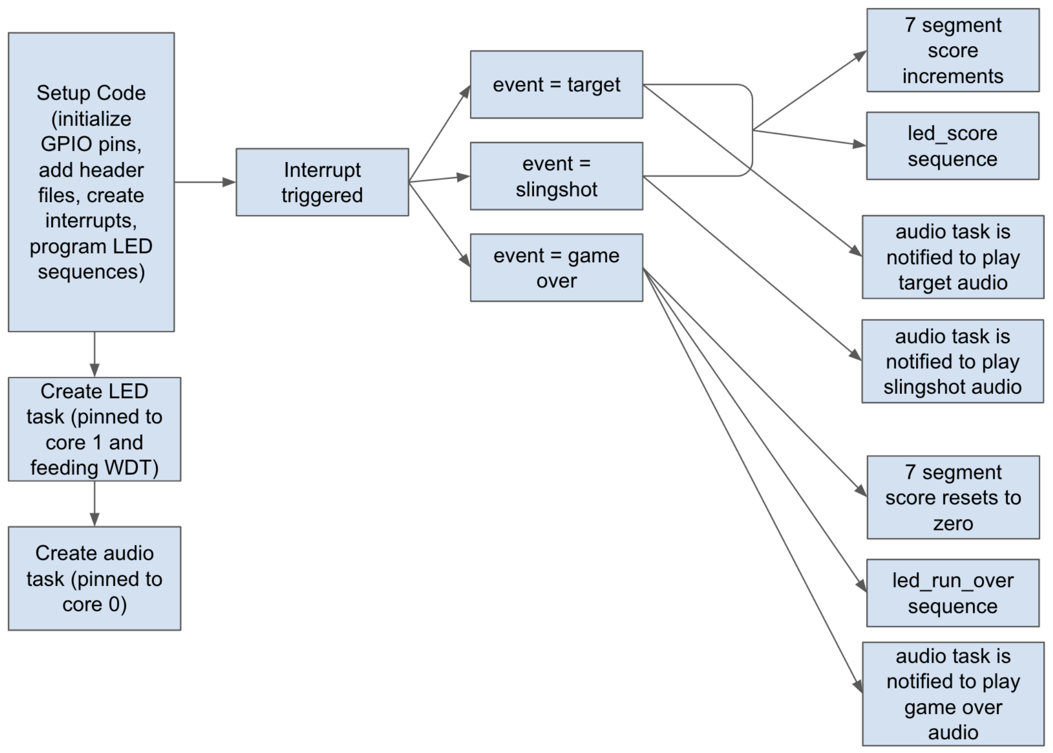

Setup Code (initialize GPIO pins, add header files, create interrupts, program LED sequences)
Create LED task (pinned to core 1 and feeding WDT)
Create audio task (pinned to core 0)
Interrupt triggered
event = target
event = slingshot
event = game over
7 segment score increments
led_score sequence
audio task is notified to play target audio
audio task is notified to play slingshot audio
7 segment score resets to zero
led_run_over sequence
audio task is notified to play game over audio

Figure 34. Software Logic Flow Diagram.

4 System Integration Testing

4.1 How the Integrated Set of Subsystems was Tested.

The testing process for the Tabletop Pinball Machine was extensive due to the sheer number of subsystems that needed to be tested and fully functional. First, every subsystem was tested separately with the board in order to ensure that the wiring of the board, as well as the code, was functioning properly. The flippers and slingshots were both tested in similar methods using limit switches and buttons that would complete the circuit with the 12 V power supply. The limit switches needed to be configured particularly for the uses in the Pinball Machine where pressing down on the switch would trigger a falling edge for the ESP-32 to then enact an interrupt. Preliminary testing showed that the trigger would cause the solenoids for the flippers and slingshots to engage once the switches were pressed. The same followed for the targets, as they used limit switches as well to trigger a falling edge for the ESP-32 to detect and respond to. For the gutter sensor, all that was needed to confirm its functionality was if the beam could be detected and then broken, which was also indicated by a falling edge signal. The audio was tested by hooking up the LRC, BCLK, DIN, GND, and Vin pins to the ESP-32 with the 3.3 V power rail while the positive and negative terminals of the speaker were wired to the positive and negative junctions on the I2S Amp Breakout PCB. Lastly, the 7-segment display was tested in a similar manner, by hooking up the 5 pins (5V, VDD, SCL, SDA, and GND) to their respective pins on the ESP-32 and testing its response to a target press.

Bigger challenges surfaced once it came time to implement the subsystems together. Since there were multiple subsystems operating on different power sources, we decided to slowly and methodically increment the amount of subsystems operating at the same time. First, we wired up the audio subsystem to our powered board. At this time, the code was written to have the audio system play ambient music to the tune of the *Notre Dame Victory March*. Having ambient music play any time the system was powered was a good indicator if other systems were working properly or if the board was malfunctioning.

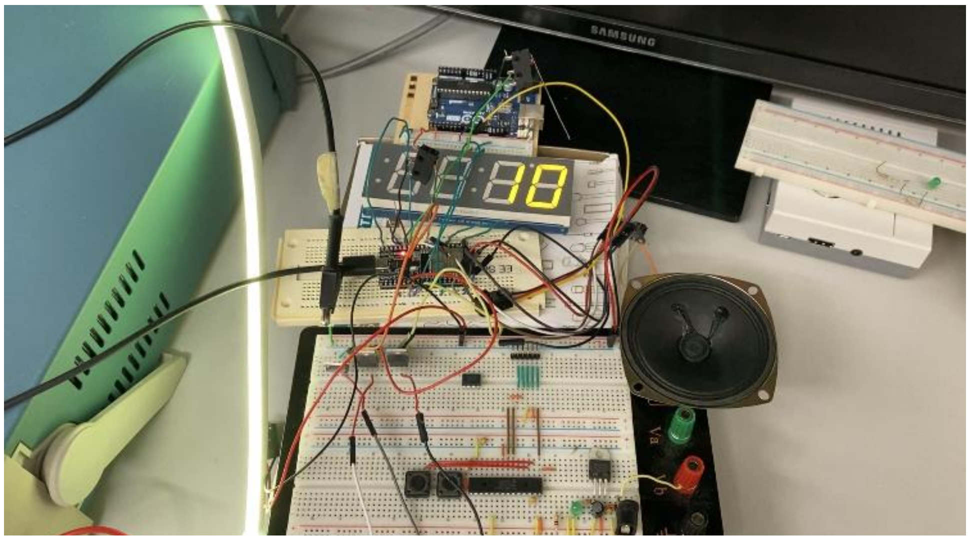


Figure 35. DevKit System Integration Testing Setup: 7-Segment, Audio, and Small Switches.

After setting up the audio, the 7-segment display was next. Since there was no method of scoring yet, the key indicator of proper integration was for the displays to have written values on them, which came to be “1” and “0”. Once the targets were added to the system, they functioned properly by causing the 7-segment display to increment by 5 points every time the target’s limit switch was pushed. Another sign of the system still working as expected was the speaker emitting the designated sound for the target being triggered. The same followed for the solenoids of the slingshots–once the solenoid was pulled, the score incremented by 5 and the speaker emitted a sound for the slingshot being activated. Ultimately, these solenoids were not used for scoring, but this was a way to confirm the limit switches were triggering interrupts. The activation switches for the solenoids were not in use yet because the solenoids were not yet connected to the 12 V power supply. Setting up the LED posed no issues. Once the RGB pins and the 12 V power supply were configured, the subsystem worked without a problem, flashing with blue, yellow, and green ambient lighting with the alternate routines appearing once a target or slingshot was triggered. The last subsystem to integrate was the flippers, which worked after resolving some of the stiffness in the mechanical part of the design.

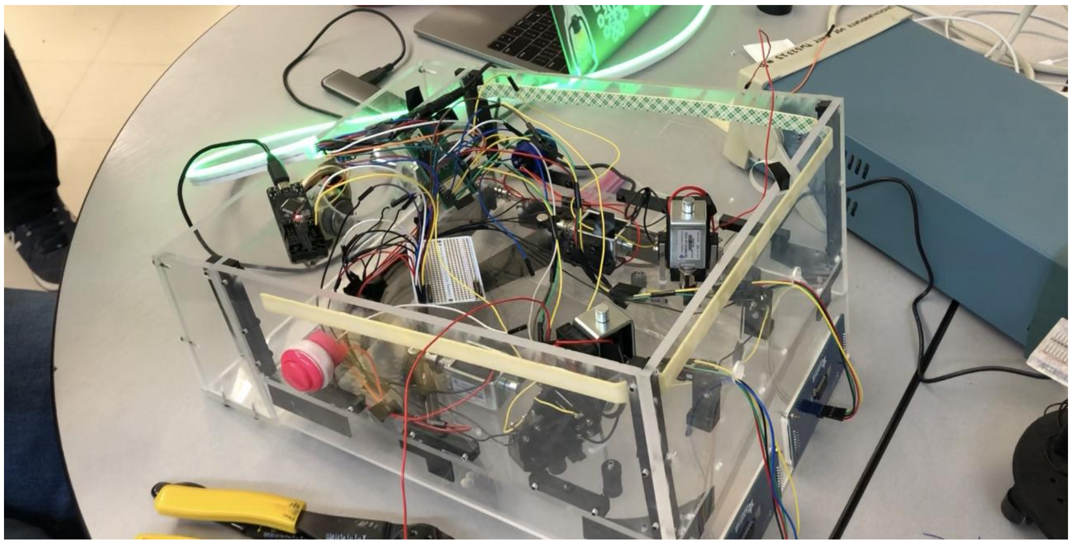


Figure 36. System Integration Testing Setup.

Most of the problems during the system integration came from three issues: processing time, improper wiring, and power consumption. Processing speeds were easy to fix using the dual core functioning on the ESP-32 to divide tasks onto two cores instead of one. This offloaded the demands that one core would have to run, while giving the appearance of faster processing speeds. Fixing the issues with power was more complex. After all of the subsystems were properly integrated, the team hooked up each subsystem to its proper power source from the designed board. Immediately something went wrong once the system was turned on because the audio sputtered and parts of the board were hot to the touch.

A few more tests were performed, but then other subsystems started malfunctioning as well or would not even power on. Checking each junction on the board revealed that there was a short between one the 3.3 V rail and ground. As this short did not exist during previous testing of the assembled board, we surmised that something had fused internally. Since this issue came about after adding in the limit switches into the system for the slingshot solenoids, a closer look revealed that the slingshots were wired such that they were normally shorted between 12 V and ground. This led to us rewiring the target limit switches entirely, and testing them more thoroughly with the multimeter.

Once a new board was soldered and better solder connections made on our extension wires, the system worked aside from one final problem involving the solenoids. Since the solenoids needed high amounts of power, and they were in close proximity to many other wires hooked up to other subsystems, and any time the solenoids were activated there would be a collapse in the surrounding magnetic field. This then affected other wires, triggering interrupts to take place that should not have occurred. Adding the 1N4004 high-resistance rectifier diodes for each of the solenoids prevented current from flowing in the opposite direction, and contained the magnetic field breakdown. The whole system could then be placed right side up, and could be tested with the ball and the plunger. And the only modification the plunger system needed was some filing down of the opening for its spring launcher to move smoothly through.

4.2 Demonstrating that the Overall System meets the Design Requirements

As explained in Section 4.1 and previous subsystem descriptions in Section 3, despite needing to solder a new board and rewire the subsystem and board connections, each subsystem worked to the best of its ability. The targets, upon their limit switches being pressed, would trigger the audio to play the target sound and 7-segment to display the incremented score. The slingshot limit switches would activate the solenoid and push back against the ball without creating a short-circuit. When starting the game, the plunger could be pulled back and launch the ball with ease. Once the ball would roll past the gutter sensor and break the beam, the 7-segment display would reset to 0, and the audio system would play the “game over” sound file, with the LED’s following suit. Since the audio system would play each time the target or gutter sensor was triggered, the audio successfully met its design requirements. At any time during the game, the user could push the flipper buttons and they would activate the flippers, which satisfied their design requirements. All subsystems were functional and performed as they were designed, and the machine was a playable pinball machine.

5 Users Manual/Installation manual

5.1 How to Install

Installation is minimal for the desktop pinball machine. Constructed for college students to place on their dorm room tables, ease of installation was a key design consideration. The user needs a flat surface to rest the machine on, ideally with enough surface friction to prevent the pinball machine from shifting around when the plunger is pulled to launch the ball. Additionally, the machine should be situated near a wall outlet or power strip to easily power the machine.

5.2 Setup

To set up the desktop pinball machine, the user needs to place the pinball at the top of the plunger's spring. The 12 V power supply will need to be plugged in to power the machine.

5.3 How to tell if the Pinball Machine is Working

Once powering up the pinball machine, the user should see the ambient LED light sequence and the two 7-Segment displays light up: the left-hand one indicates the score, which should print the number zero upon start-up, and the other prints a message, currently "NDEE." The user can test the two flipper buttons on the side of the chassis to test if the flippers on the playing field will kick out. Once the pinball is launched using the mechanical plunger, the user can confirm if the score is incrementing and a sound effect is played when the targets and slingshots are struck by the ball during a live game. Further, the score should reset to zero and play a corresponding audio file when the game ends and the ball rolls through the gutter sensor.

5.4 How the User can Troubleshoot the Machine

If the pinball machine's features are not lighting up, the user should confirm that the machine's power source is fully plugged into the wall. If the flippers are not operating as expected, the user should inspect the screws holding the flippers into the playing field and tighten or loosen them if necessary. If the audio track does not play and/or the score does not update on the 7-Segment display, the user should ensure that each of the subsystem's wires are plugged into the correct corresponding pin on the PCB and that there are not any loose wires.

6 To-Market Design Changes

Several design changes should be made to ensure the desktop pinball machine is suitable to be sold commercially. We would add a piece of acrylic to the bottom and top of the machine to fully enclose the wires and playing field. The PCB board would be mounted onto the bottom or side of the inner chassis such that it doesn't get jostled around when the user is playing a game. The channel at the bottom of the playing field where the ball rolls after a game has ended would be designed as a separate component with a slant to naturally roll the pinball back to its starting position at the base of the plunger. To ensure that the targets and slingshots can be triggered, a custom made pinball would be designed for a commercial product. This ball needs to be about 20 mm in diameter and 10 g in mass. Additional sound effects and LED patterns would be added to enhance the variety of the player's experience. The playing field can have different themes to appeal to individual taste. Colors of the physical components can be personalized and differentiate one user's desktop pinball machine from another. Designs can be placed on the targets and slingshots, which will have scoring limit switches added as well. Additional obstacles can be incorporated to increase the complexity of the game, and the slingshots can be positioned to be struck more frequently. As the prototype machine stands now, there are several areas where the ball can get stuck. The ramp, targets, and flippers would be repositioned to ensure the ball can move freely more effectively throughout the playing field.

7 Conclusion

We developed and built a pinball machine that fits on a standard Notre Dame dorm room desktop, is carryable by a student, and can be powered by a standard wall supply. It is interactive and catches the attention of users and passersby alike. It is in need of refinement, most especially requiring a heavier ball, tighter limit switches for the targets, and a sloped lower playing field such that the ball properly rolls into the gutter. Despite this, we have demonstrated a significant amount of electromechanical and software integration, and have received positive feedback from fellow Notre Dame students, the prospective customers if this were to become a commercial product. Electrically and software-wise, our machine fulfills all of its stated design requirements, and we consider our project to have broadly fulfilled its goals.

8 Appendices

8.1 Datasheet Links

Microcontroller (ESP-32-WROOM-32E-N8) datasheet
LED strip (iPixel non-addressable flexible LED strip) datasheet
Power NFET (IRLB8721) datasheet
Push-Pull Solenoid Technical Information
12 V Wall Power Supply Technical Information
7-Segment Display and I2C Backpack Technical Information
7-Segment Backpack Schematics
Speaker Technical Information
Audio Amplifier Datasheet

8.2 Board Schematic and Layout

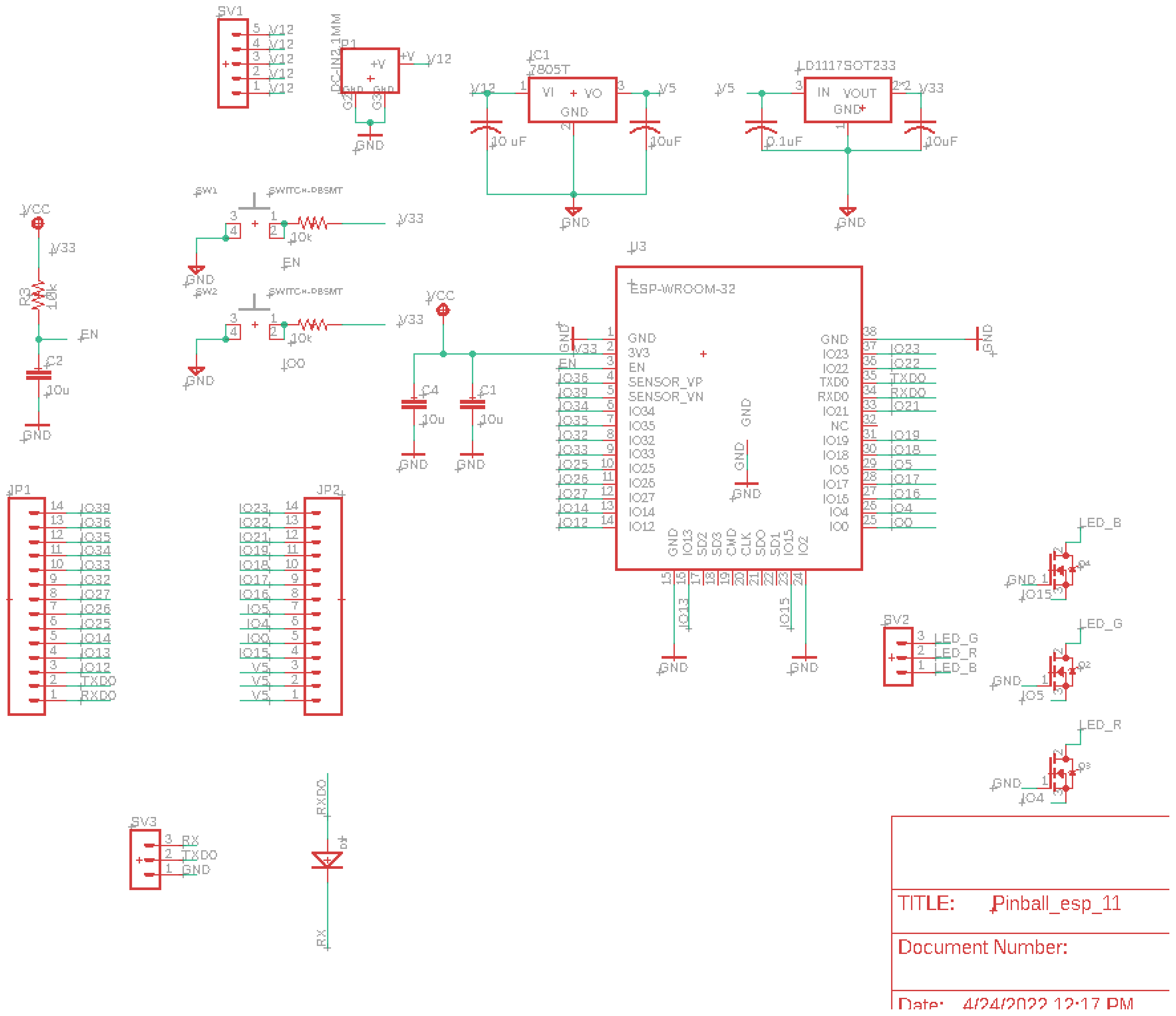


Figure A1. Board Schematic.

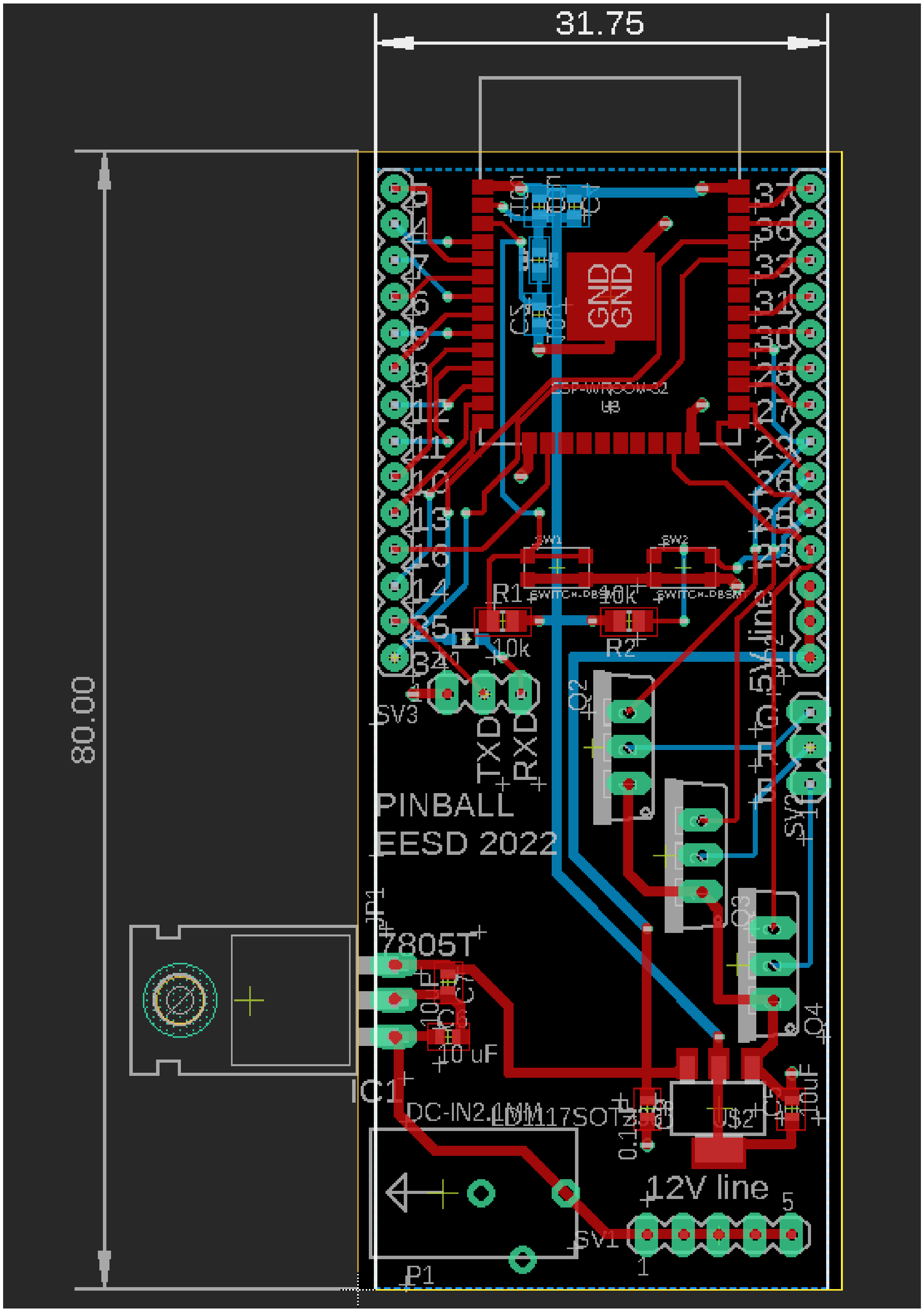


Figure A2. Board Layout.

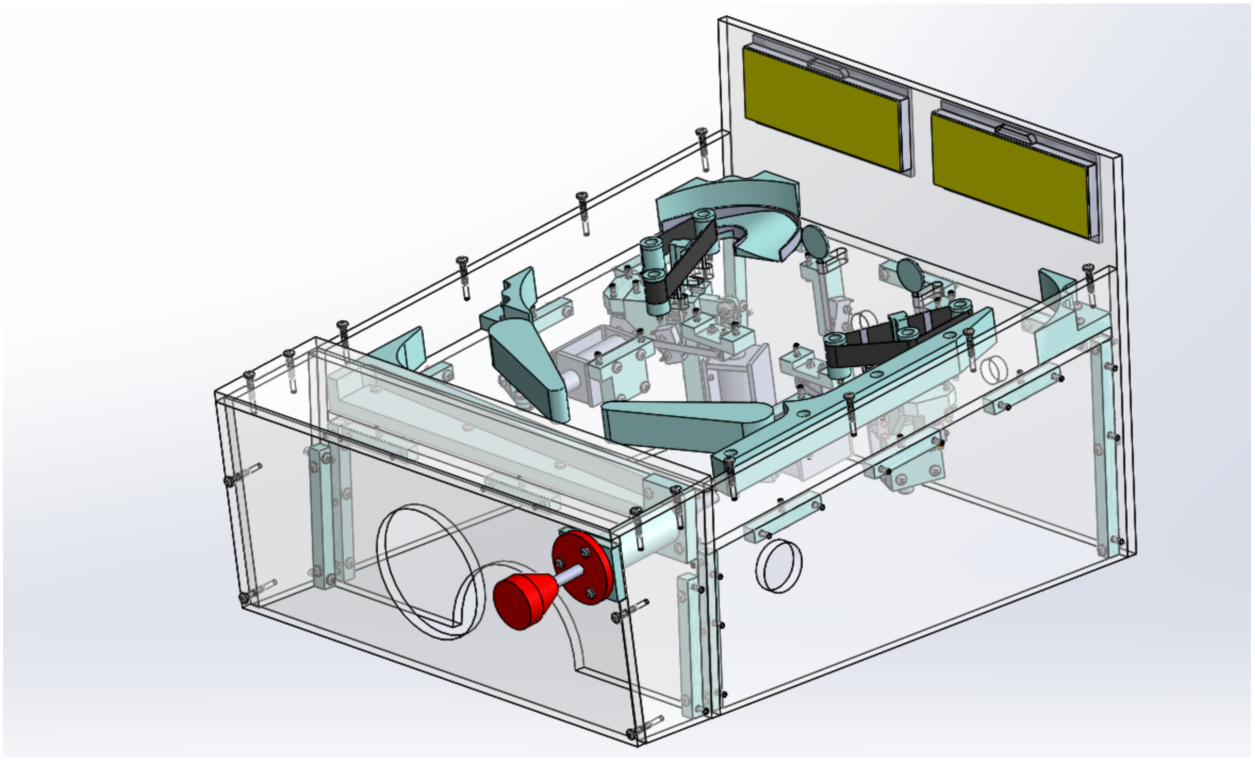


Figure A3. CAD Model, front.

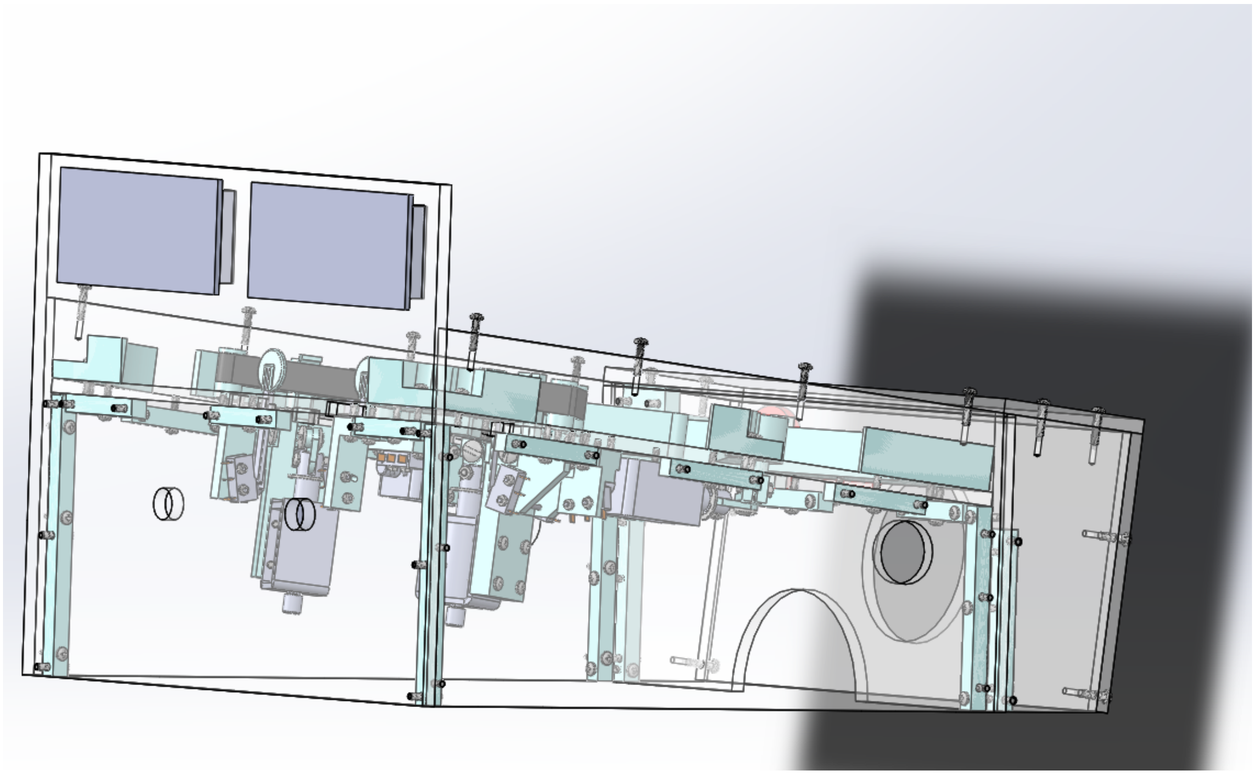


Figure A4. CAD Model, back.

8.3 Complete Code Listing

(Note that this section does not include audio header files as they are hex digits and exceptionally long, while not shedding further light on our project. They can be found on our website.)

```
/* EE 41440: Senior Design II
   Pinball 2022
   University of Notre Dame

   This code runs the 2022 Pinball Senior Design Project. It is written for an ESP32
WROOM Module running
   on a custom-designed PCB board. The microcontroller is responsible for keeping
track of the score and
   game number, and reporting them on a seven-segment display, as well as
determining "events" (scoring
   and the end of the game). When events occur, the microcontroller will update the
display, flash an
   LED strip, and play audio, as appropriate. In the absence of events, the
microcontroller will provide
   ambience in the form of audio and an LED routine.

   Team website:
http://seniordesign.ee.nd.edu/2022/Design%20Teams/Pinball/top_page.html

   Team Members:
     Alisa Nguyen
     Lauren Rymza
     Victoria Smerdon
     Yadviga Tischenko
     James Venditto

   This code is intended for an ESP32 WROOM Module.

   Code authors and contributors:
     Alisa Nguyen
     Lauren Rymza
     James Venditto

    Audio Library courtesy of Earle Philhower and Alex Wende
(https://github.com/earlephilhower/ESP8266Audio)
    Multicore Code example courtesy of Rui Santos
(https://randomnerdtutorials.com/esp32-dual-core-arduino-ide/)

*/

////// Global Definitions:
```

```
// Pins:
#define GAME_OVER_PIN 12                // game over pin (beam-break sensor)
#define TARGET_PIN 16                 // target pin
#define SLINGSHOT_PIN_1 13              // slingshot pin
#define SLINGSHOT_PIN_2 19              // slingshot pin

// Logic Definitions:
#define GAME_OVER 1
#define TARGET 2
#define SLINGSHOT 3

// Global Variables:
unsigned int score = 0;              // score variable


////// Definitions for LED:
#define BLUE 15
#define RED 4
#define GREEN 5


////// Definitions and headers for 7-Segment Display:
#include <Wire.h>
#include <Adafruit_GFX.h>
#include "Adafruit_LEDBackpack.h"

// Score Matrix:
TwoWire I2CMatrix_score = TwoWire(0);
Adafruit_7segment matrix_score = Adafruit_7segment();

// Game Number Matrix:
TwoWire I2CMatrix_game = TwoWire(1);
Adafruit_7segment matrix_game = Adafruit_7segment();


////// Definitions and headers for Audio:
#include <Arduino.h>
#include "AudioGeneratorAAC.h"
#include "AudioOutputI2S.h"
#include "AudioFileSourcePROGMEM.h"
#include "target.h"
#include "gameOver.h"
#include "slingshot.h"

// Headers for Watchdog Timer:
#include "soc/timer_group_struct.h"
```

```
#include "soc/timer_group_reg.h"

// Initialize ESP8266 Audio Library classes:
AudioFileSourcePROGMEM *file_target;
AudioFileSourcePROGMEM *file_gameOver;
AudioFileSourcePROGMEM *file_slingshot;
AudioFileSourcePROGMEM *file_ambient;
AudioGeneratorAAC *aac;
AudioOutputI2S *out;
volatile bool playing = 0;
volatile byte loadTrack = 0;      // for audio
volatile byte event = 0;          // for main loop


////// External Interrupt function with software switch debounce:
void IRAM_ATTR handleInterrupt(){

  static unsigned long last_interrupt_time = 0;
  unsigned long interrupt_time = millis();

  // If interrupts come faster than 200ms, assume it's a bounce and ignore:
  if (interrupt_time - last_interrupt_time > 200){
   // Determine which switch was triggered, and which track to play:
    if(!digitalRead(GAME_OVER_PIN)){
      loadTrack = GAME_OVER;
      event = GAME_OVER;
      score = 0;
    }

    else if(!digitalRead(TARGET_PIN)){
      loadTrack = TARGET;
      event = TARGET;
      score = score + 5;
    }

    else if(!digitalRead(SLINGSHOT_PIN_1)){
      loadTrack = SLINGSHOT;
      event = SLINGSHOT;
      score = score + 5;
    }

    else if(!digitalRead(SLINGSHOT_PIN_2)){
      loadTrack = SLINGSHOT;
      event = SLINGSHOT;
      score = score + 5;
    }
    playing = 1;
  }

  last_interrupt_time = interrupt_time;
}

void setup() {

////// Pin Setup:
// Configure trigger pins to inputs with internal pull-up resistors enabled:
  pinMode(GAME_OVER_PIN,INPUT_PULLUP);
  pinMode(TARGET_PIN,INPUT_PULLUP);
  pinMode(SLINGSHOT_PIN_1,INPUT_PULLUP);
  pinMode(SLINGSHOT_PIN_2,INPUT_PULLUP);

// Create interrupts for each trigger:
  attachInterrupt(digitalPinToInterrupt(GAME_OVER_PIN),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(TARGET_PIN),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(SLINGSHOT_PIN_1),handleInterrupt,FALLING);
  attachInterrupt(digitalPinToInterrupt(SLINGSHOT_PIN_2),handleInterrupt,FALLING);

////// Audio Setup:
  out = new AudioOutputI2S();
  aac = new AudioGeneratorAAC();
  out -> SetGain(0.25);          // Set the volume

  xTaskCreatePinnedToCore(
                  AudioTask,    // Task
                  "AudioTask",  // Task name string
                  10000,
                  NULL,
                  1,            // Priority
                  NULL,
                  0);           // Use core 0
  delay(500);

////// LED Setup:
  pinMode(BLUE, OUTPUT);
  pinMode(RED, OUTPUT);
  pinMode(GREEN, OUTPUT);

  xTaskCreatePinnedToCore(
                  LEDTask,
                  "LEDTask",
                  10000,
                NULL,
                1,
                NULL,
                1);          // Use core 1
  delay(500);


////// 7-segment Setup:
  #ifndef __AVR_ATtiny85__
  #endif
// Score Display:
  I2CMatrix_score.begin(33,32);  // V_IO is 5 volts, SDA, SCL
  matrix_score.begin(0x70, &I2CMatrix_score);
  matrix_score.clear();
  matrix_score.print(0, DEC);
  matrix_score.writeDisplay();

// Game Number Display:
  I2CMatrix_game.begin(23,21);   // V_IO is 5 volts, SDA, SCL
  matrix_game.begin(0x70, &I2CMatrix_game);
  matrix_game.clear();
  matrix_game.print("NDEE");
  matrix_game.writeDisplay();

}

////// LED Task:
void LEDTask( void * pvParameters ){

  for(;;){
    led_ambient();
    TIMERG0.wdt_wprotect=TIMG_WDT_WKEY_VALUE;    // shutoff watchdog timer in
core
    TIMERG0.wdt_feed=1;
    TIMERG0.wdt_wprotect=0;
  }
}


//// Audio Task:
void AudioTask( void * pvParameters ){

  for(;;){
    // Stop the current track if playing
    if(playing && aac->isRunning()) aac->stop();
```

```
// Choose event based on what was triggered:
if(loadTrack == GAME_OVER){
  file_gameOver = new AudioFileSourcePROGMEM(gameOver, sizeof(gameOver));
  aac -> begin(file_gameOver,out); // Start playing the track loaded

  playing = 0;
  while(playing == 0){
    if (aac->isRunning()) {
      aac->loop();
    }
    else{
      playing = 1;
      loadTrack = 0;  // don't play any sounds twice
    }
  }

}

else if(loadTrack == TARGET){
  file_target = new AudioFileSourcePROGMEM(target, sizeof(target));
  aac -> begin(file_target,out); // Start playing the boost sound
  playing = 0;
  while(playing == 0){
    if (aac->isRunning()) {
      aac->loop();
    }
    else{
      playing = 1;
      loadTrack = 0;  // don't play any sounds twice
    }
  }

}

else if(loadTrack == SLINGSHOT){
  file_slingshot = new AudioFileSourcePROGMEM(slingshot, sizeof(slingshot));
  aac -> begin(file_slingshot,out); //Start playing the track loaded
  playing = 0;
  while(playing == 0){
    if (aac->isRunning()) {
      aac->loop();
    }
    else{
      playing = 1;
      loadTrack = 0;  // don't play any sounds twice
    }
```

```
      }

    }

    else{

    } //end of ELSE statement
  }
}


////// Main Loop:
void loop() {
// Interrupt has triggered an event:
  if(event){

    // Choose event based on what was triggered:
    if(event == GAME_OVER){
      matrix_score.print(0, DEC);
      matrix_score.writeDisplay();
      led_game_over();
    }

    else if(event == TARGET){
      matrix_score.print(score, DEC);
      matrix_score.writeDisplay();
      led_score();
    }

    else if(event == SLINGSHOT){
      matrix_score.print(score, DEC);
      matrix_score.writeDisplay();
      led_score();
    }

    event = 0;              // Event is over: reset
  }

// Nothing is currently happening:
  else{
  }

}

////// LED Color Functions:
void rgb_off(){
  digitalWrite(RED, LOW);
  digitalWrite(BLUE, LOW);
  digitalWrite(GREEN, LOW);
}

void rgb_red(){
  digitalWrite(RED, HIGH);
  digitalWrite(BLUE, LOW);
  digitalWrite(GREEN, LOW);
}

void rgb_blue(){
  digitalWrite(RED, LOW);
  digitalWrite(BLUE, HIGH);
  digitalWrite(GREEN, LOW);
}

void rgb_green(){
  digitalWrite(RED, LOW);
  digitalWrite(BLUE, LOW);
  digitalWrite(GREEN, HIGH);
}

void rgb_magenta(){
  digitalWrite(RED, HIGH);
  digitalWrite(BLUE, HIGH);
  digitalWrite(GREEN, LOW);
}

void rgb_yellow(){
  digitalWrite(RED, HIGH);
  digitalWrite(BLUE, LOW);
  digitalWrite(GREEN, HIGH);
}

void rgb_cyan(){
  digitalWrite(RED, LOW);
  digitalWrite(BLUE, HIGH);
  digitalWrite(GREEN, HIGH);
}

void led_ambient(){
  rgb_green();
  delay(430);
  rgb_blue();
  delay(430);
  rgb_yellow();
  delay(430);
}

void led_score(){
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
}

void led_game_over(){
  rgb_red();
  delay(250);
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_magenta();
  delay(250);
  rgb_magenta();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_red();
  delay(250);
  rgb_red();
  delay(250);
  rgb_blue();
  delay(250);
  rgb_magenta();
  delay(250);
  rgb_off();
}
```