Szander Brenner
4/23/2025
99974691

EE Design 1 Technical Report
Final Project
"Bubble Hockey Scoreboard"

Contents

Introduction

I recently acquired a Bubble Hockey table. It is not a complex and expensive version with a button-deployable puck, but it does come with a scoreboard that tracks goals and game state. However, this scoreboard was broken. The goal of my system was to replace this scoreboard and recover the functionality from the previous version. This meant it needed to have a sensor that can automatically track goals. It needed a speaker that could play sounds with the goals and with the game ending. In addition, it needed some indicator of current goals and game time.

Ideally, this system would be small enough to be mounted to the bubble hockey arena and be powered from a 9V battery. In completion, all these goals were met, but there is room for improvement in the next revision.

Methods

Requirements

In this section, I will discuss why I chose the components I did to fulfill the requirements. For this project, we needed an analog input, analog output, digital input, digital output, an analog signal amplified to a speaker, system status displayed on an LCD, and some sort of power regulation.

Since I was designing a scoreboard, I knew that the speaker output would come from a goal score and game ending like in the actual sport. The LCD would display all the game information like game time and score. Furthermore, the sensor most used for detecting if a goal was scored is the beam break sensor. This is the same device used in garage doors to detect if something is blocking the garage door path. This is a digital input as described in its datasheet (Digikey). This takes care of the digital input, analog output and LCD requirements.

The only things left to decide are digital output, and analog input. I originally wanted to have seven segment displays display the number for score and time, but I realized that with that configuration I couldn't just do GPIO mapping, I would need a special seven segment driver chip to control 8 seven segments. Instead, for simplicity I just used five LEDs for each side to track goals if we are playing a game to either 3 or 5 goals. For the analog input, I decided to use a potentiometer to be a 'state selector'. You can scroll through the different game options depending on the voltage on the output. Lastly, I went with the switching regulator as my power regulator. While it does take up significantly more space on the PCB, its efficiency was better. This was found in a previous lab. Since I wanted this to be battery powered, I could sacrifice that space for longer lasting batteries. With all my components selected, I can begin researching more into the components and designing the system.

Analog Input – Potentiometer

A typical Trimpot potentiometer has a pinout like the one seen to the right (GeeksforGeeks, 2021). There are two fixed ends that connect to either side of the total resistive material. The middle wiper pin can be spun using a dial to connect to any part of the potentiometer. From Ohm's Law we know that V = I*R. Resistance (R) is equal to pL/A where p is rho for resistivity of the element and A is the cross-sectional area of the wire. In a potentiometer or rheostat, the cross-sectional area is kept constant, and the materials resistivity is kept constant. Therefore, we can modulate the L of the material by connecting the wiper to various points in the material, effectively changing the resistance. These potentiometers are used in a wide range of applications, but most notably audio systems for volume modulation.

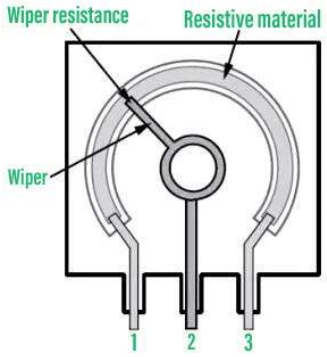

Figure 1: Potentiometer Pinout (GeekforGeeks, 2021)

Digital Input – Beam Break Sensor

The beam break sensor is the real neat part of this project. The concept is in a lot of common places, most notably garage doors (RealPars.). There are also multiple different styles of photoelectric sensors. The main three are the through-beam (beam break), retroreflective, and diffused. The image below shows their setups.

Figure 2: Types of Photoelectric Sensors(RealPars.)

They all operate on the same principle. The differences are that the through-beam requires a separate emitter and reciever to send and catch the light, the retroreflective beam has a mirror to reflect it, so the emitter and reciever are the same element, and the diffused sends a beam and when some object reflects the light back, that counts as a trigger. In each of the setups, a beam of IR light is sent to a reciever. The reciever gets hit by the beam, which relies on the photoelectric effect. The IR light excites electrons in the material, generating a measurable current. In the first two configurations, the through-beam and retroreflective, the 'normal' state is continuous current. When the current stops, there is something in the beams path, triggering the sensor to output either a high signal or low signal

depending on the manufacturer specifications. The diffused works opposite to this. There is no received beam until something moves in its path and reflects the light. In my system, I am using the through beam to detect when a puck passes into the goal. The signal is sent to my microcontroller for processing.

DAC to Audio Amplifier

DAC

The last complex aspect of this project is to create the audio output for the speaker. We will be outputting a signal to a DAC since our Pico does not have a built in DAC. The DAC will output a voltage to an audio amplifier for a signal that can drive a speaker. For a quick background, DACs take in digital value and produce an analog signal out, typically 0-5V. Our DAC, the LTC1661, takes 2.7V-5.5V VCC, which means that it can provide output voltage from 0-5V (Analog Devices). Our chip is programmed through SPI. It has no MISO pin, meaning no data can be read from the device. However, there are only 3 operations we have for each channel. Load, Wake, and Load and Wake. Our DAC has a 10-bit resolution, meaning that we have 1024 steps from 0V to our Vref.

How do DACs work?

There are a few different types of DACs. There are Resistor String DACs, Binary Weighted, R-2R Ladder, and Sigma-Delta DACs. Each has some benefits and tradeoffs. Resistor Strings are slow and require lots of resistors for higher resolution. Binary Weighted uses resistors with values weighted by powers of 2, but they require more precise resistor values. R-2R are very common and use only two resistor values (R and 2R) in a ladder configuration, reducing complexity while maintaining precision. Sigma-Delta DACs are more expensive and higher quality, usually found in audio systems (OpenAI, 03/2025).

Our DAC is a resistor string architecture. This allows rail-to-rail output voltages and are typically found in low power, small scale DACs like our LTC1661. They only draw around 60 uA per DAC (there are two on the chip).

Where are DACs used?

DACs are used all around, but most widely used in audio systems. Our songs and software are all ones and zeros, but if we wanted to create a signal that we can hear, we need to transfer it into an analog signal. This is where DACs are used.

Not only that, but last week we saw that DACs are used in ADCs too. Using successive approximation register ADCs, they use a DAC to output a voltage to compare to. This makes them very important to applications requiring ADCs, which are about any monitoring system or program with a sensor.

Amplifiers

Op-amps were developed in the 1930s-1950s to conduct analog computing for mathematical expressions like adding, subtracting, and multiplying (OpenAI, 2025). They are also used for easy amplification of analog signals for communications and measurement systems. Due to their high gain and overall versatility, they have many applications. To customize them to meet system requirements, the *feedback* is set by connecting the output to one of the input terminals with a direct connection or through other electronic components like capacitors and resistors.

Their layout typically consists of 3 signal terminals and two voltage (for power) terminals. A diagram is shown on the right. The two inputs are typically labelled inverting and non-inverting input. This is because typically if your input signal is connected to the inverting input, the output will invert the level of voltage. For example, 5V in goes -5V out depending on the input power rails. Both inputs are very high impedance and therefore draw very little current from the input. The output can both source and sink current, meaning that they can provide current/voltage and also take excess to ground (Storr, 2024). However, for most op-amps, this is on the order of milliamps. This introduces the need for differentiation between general op amps and audio amplifiers which are designed to drive low impedance loads like speakers and provide high current.

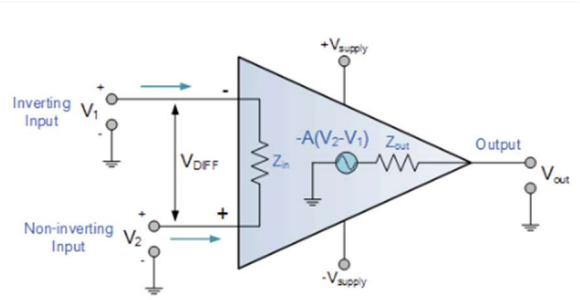

Figure 3: Op amp general image, (Storr, 2024)

There are many ways to choose an op-amp for the circuit you are designing. The main metrics to focus on are Rail-to-Rail, Gain Bandwidth product, slew rate, operating voltage, max operating temperature, operating supply current, common mode rejection ratio, input referred noise, and mounting style (ECE3923C Design 1). Rail-to rail specifies that the output can (or cannot) swing to and from the voltages you provide for power. This is helpful when you know what voltage output you need, and whether you need full range or not. Gain bandwidth product is important for high frequency circuits. This selects the range of frequencies that the op amp can handle without breakdown. Slew rate is the speed at which output can change. Operating voltage is always important because it sets what power supply you need. Max operating temperature is important if you are in harsh conditions. Operating supply current specifies how much current the op amp requires, so is important in low-power settings. Common-mode rejection is how much noise common to both input signals is rejected. Input referred noise is how much noise is added to the input signal through the op amp. Lastly, mounting style affects the manufacturability of your product.

For my project, I needed to use an audio amplifier because regular op amps do not drive enough current for low ohm loads like a speaker (AllAboutCircuits, 2019). The LM386 is the ideal choice for our project because it is readily available, supports wide voltage range for input, and has good documentation behind it (TI). The right shows the general schematic for a constant gain of 20 with a volume control (TI). This will effectively drive an 8 ohm speaker needed for the project.

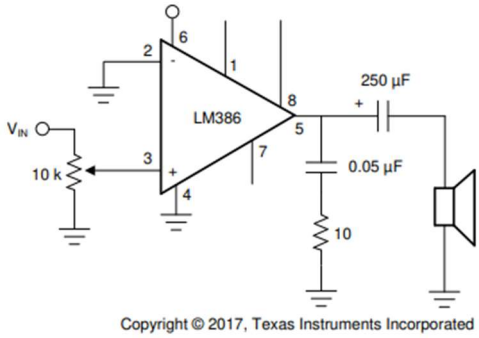

Figure 4: LM386 Schematic, (TI)

Software Block Diagram

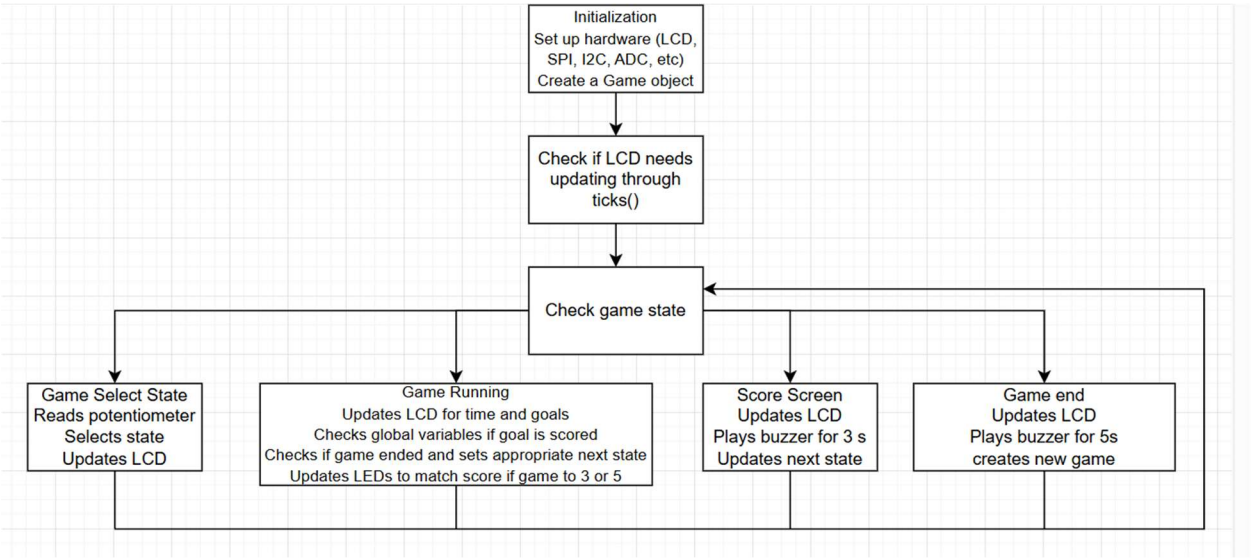

Figure 5: Software Code Diagram

The code above shows the general flow through the program. It does not explain the nuances of the code, however. I will explain how it works in this section. I split my code into 4 main files, main, game_logic, hardware, and constants. The main file organizes all the files and acts as the controller. If you are familiar with controller and datapath configuration, the game_logic is the datapath. I created a Game object that tracks all the relevant states such as current time, goals, state, etc. The main file checks this game object's current state and then calls the function corresponding to it in the game_logic file. This allows for better segmentation of code and increased readability. The hardware file is responsible for setting up all the hardware like the SPI and I2C channels. This way, my main can be more readable when I call init_DAC() and the other functions. This file also handles the interrupts. There are two main ones, button press and goal score. The interrupts both update a dictionary of flags which can be read from any file. The constants is a static file that just holds important values like pin values.

The game_logic is the brunt of the program with 180 lines of code. It is split into 4 sections corresponding to the 4 states a game could be in: Select, running, goal scored, and game over. The select state reads an ACD and based on its value, will output one of 5 options for gameplay. Once a button is pressed, it will set the next state to running and make sure the game is configured based on the selection.

In the running state, this will check whether a goal has been scored and needs to be updated. If so, it will set the next state to the goal scored state, update the counter and corresponding LED if we are playing a game to a set score, and finally check if the game has won. If it's ended, it'll branch to the game ending state instead.

In the goal scored state, it changes the LCD to say "GOAL!" and plays the buzzer for 3 seconds. It changes the next state to be the running state.

In the game over state, it assesses who won, outputs to the LCD correctly, plays the buzzer for 5 seconds, resets the LEDs and overall game state, and sets the next state equal to the select state. This completes the code loop and ensures that a new game can be played right after the end of the previous one. A decent amount of the code and structure was generated by ChatGPT, and I am referencing them here (ChatGPT, 2025).

Hardware Block Diagram

To the right shows the basic implementation of the circuit. The main components include a power regulation circuit in the form of a switching regulator, an on/off switch for the device, an LCD to display current state of the device, a potentiometer to gather analog data to select the proper state of the device, a button to select the state and move onto the running of the game, 10 total LEDs to display current goals, two beam break sensors to detect when goals are scored, and DAC, audio amplifier and a speaker to output sound from the game. This captures all the requirements of the system.

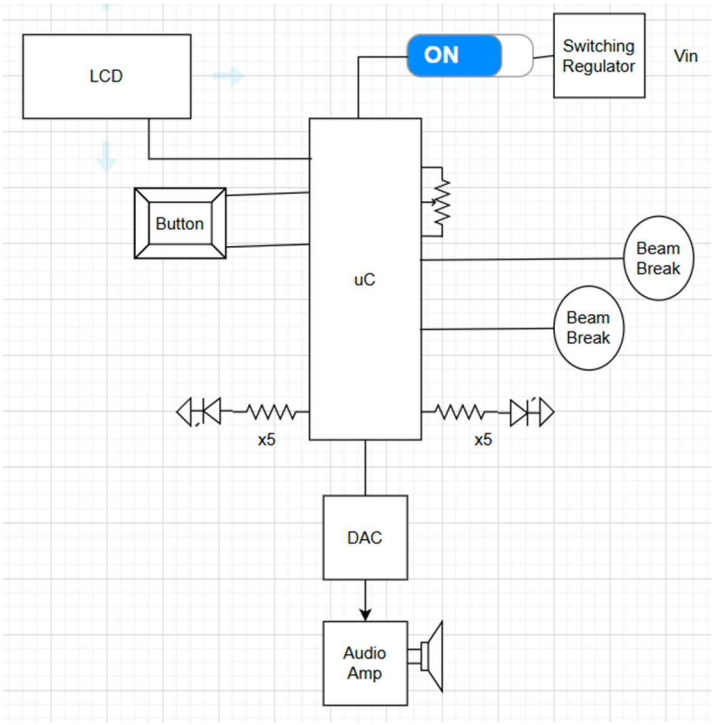

Figure 6: Hardware Block Diagram

Altium Schematic

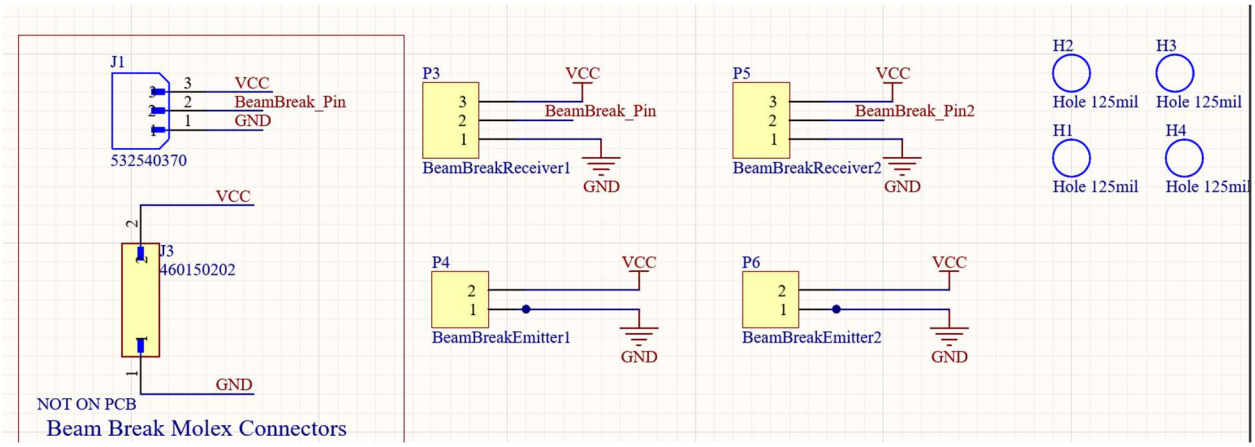

Figure 7: Header pins for Beam Break sensors.

Above shows part of the Altium Schematic that shows the through holes for mounting (on the right) as well as the headers that the beam break sensors take. The emitters just require power, and the receivers have 1 more pin for the signal. I wanted to eventually plan to replace the header pins with Molex so that it is less likely that wires fall out, so that is shown on the left.

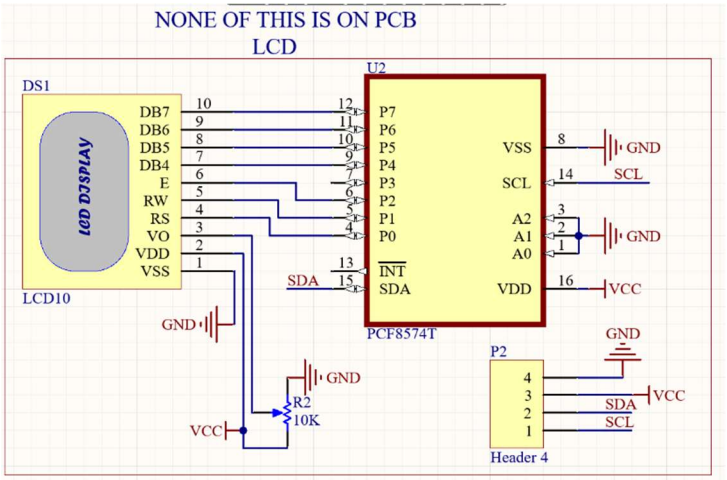

Figure 8: LCD Schematic

This part of the schematic to the left corresponds to the LCD. On the actual PCB, I just need a 4-pin header as shown on the bottom right. The left box is the actual LCD with its 16 pins out, and they connect to the I2C LCD backpack. This allows us to use I2C to control the LCD instead of the 16-pin protocol it normally has.

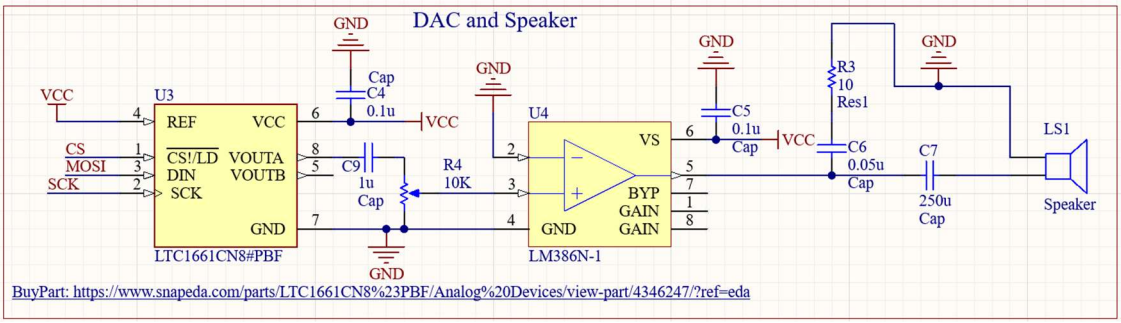

Figure 9: Speaker System Schematic

This is the schematic for the speaker system. The microcontroller sends out a signal through SPI to an external DAC. This DAC converts the message to an analog voltage, sends that to the audio amplifier which can drive low impedance loads like a speaker.

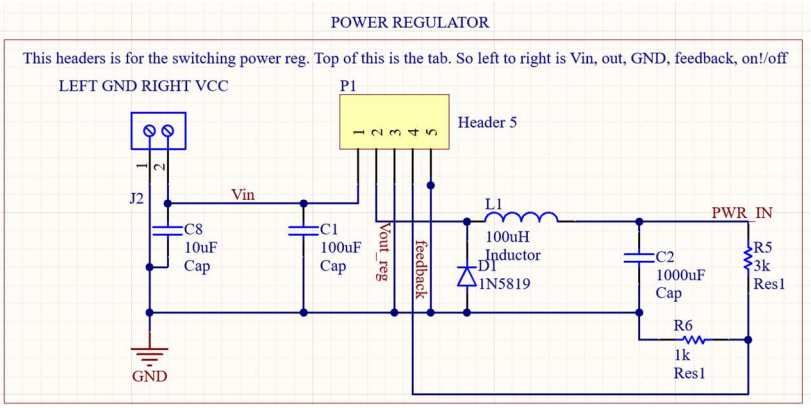

Figure 10: Power Regulation Schematic

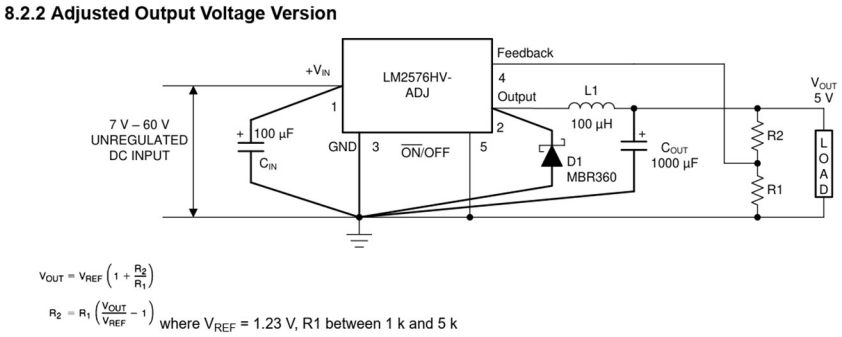

Figure 11: TI Datasheet for LM2576 Switching Regulator for adjusted output voltage (TI)

The switching regulator schematic can be found in its data sheet and mirrors this (TI). The R2 and R1 were chosen to get a 5V output and a 10uF capacitor was added to the input of the power for added signal smoothing. The rest is as the datasheet expects.

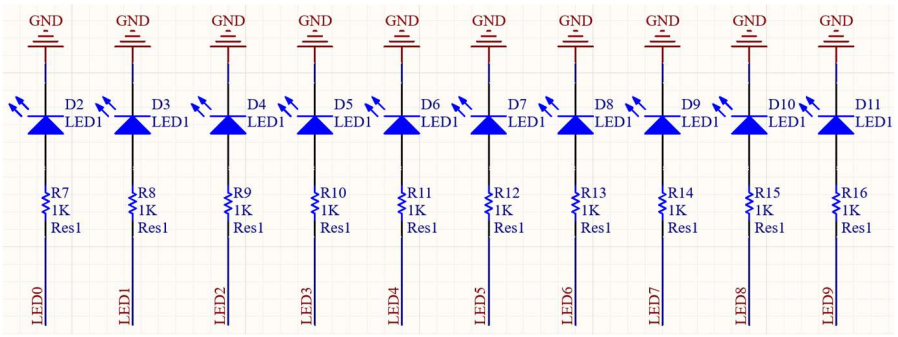

Figure 12: Simple LED Array

Above shows the basic 10 LED array (5 for each side) that represents the current goal tally. They are each connected to their own GPIO pin.

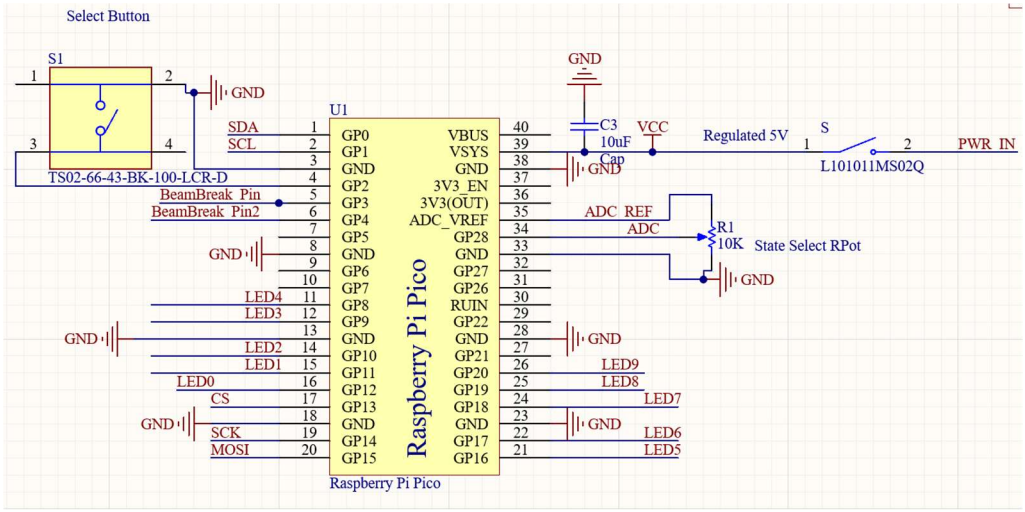

Figure 13: Microcontroller Pin Connections as well as the button, on/off switch, and state select potentiometer.

Above shows the remaining components in the schematic. The button connects to one GPIO pin and to ground. The On/Off switch connects the power regulator circuit to the rest of the board. The potentiometer connects to the ADC reference, ADC input, and GND. All the rest of the pins are labeled with net labels. The bottom has the SPI module, and the top has the LCD.

Altium PCB

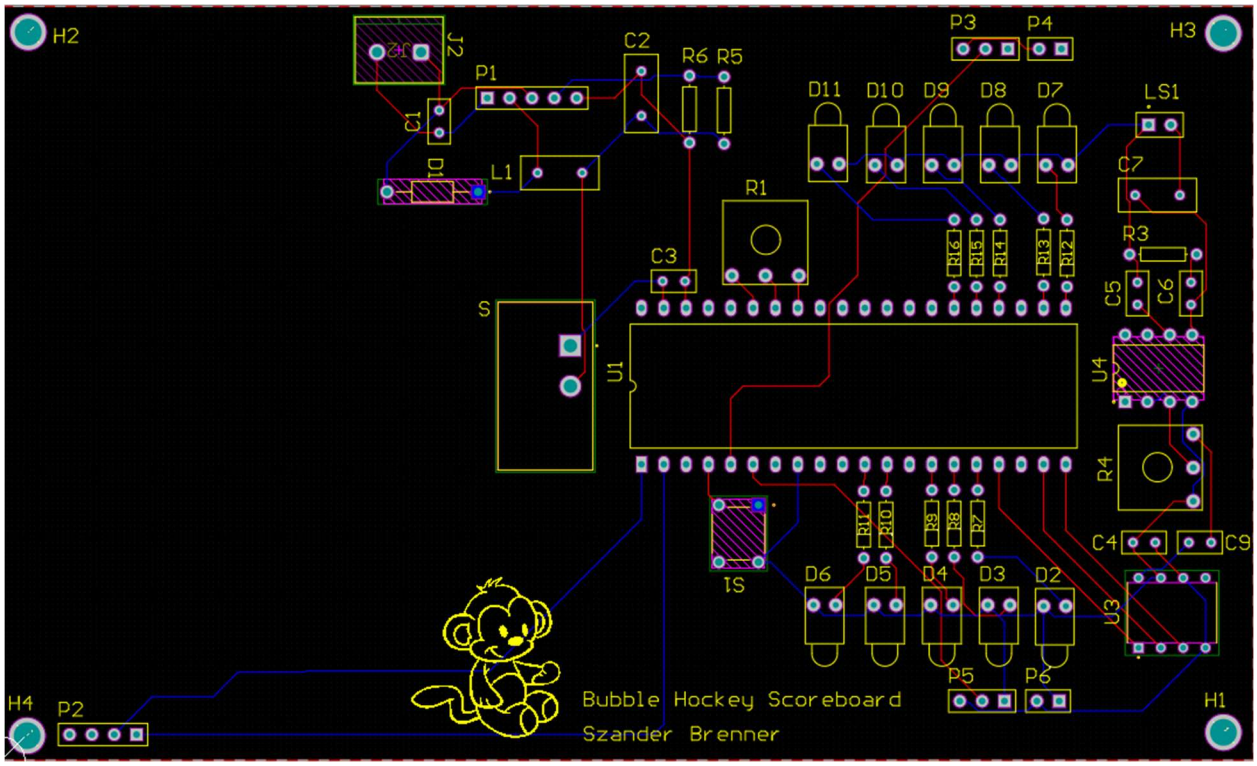

Figure 14: Altium PCB without Polygon Pours

The above image shows the layout of the board. This does not show the polygon pours. The top layer has a pour for VCC, and the bottom layer contains the ground. This allows for better signal and heat conduction throughout the entire board. If we relied on just traces, the parasitics of the trace could impact the function. The LCD sits on the left side of the board. There will be a 4-pin right angle header that allows the LCD to slide in and cover the entire top region. The power regulator sits in a 5-pin header P1 at the top. The power comes in from a screw terminal J2. Before the power gets to the microcontroller, it passes through a 4A rated switch to control if the circuit is on or off. On all the ICs and the microcontroller, notice that there is a bypass capacitor as close to the pins as possible. This smooths the power input as much as possible and means there is less interference from parasitics. The LEDs are arranged equally on both sides of the board to symbolize the two team's scores. The speaker system takes up the entire bottom of the board (right side) and eventually connects to a 2-pin header that the speaker can plug into. My main concern for this board was fitting everything in as small of an area as possible to reduce printing costs. In the end, my board was around 3 inches by 5.5 inches. I tried separating the board into two halves for the two teams to make the mechanical installation easier.

Bill of Materials

The total cost of the project's materials is around $52. The following shows the distribution.

Table1

Component	Cost	# Quantity	Link
Speaker	$8.49	1	https://www.walmart.com/ip/Unique-Bargains-2-F
Screw Terminal	$0.32	1	https://www.digikey.com/en/products/detail/w%0
Beam Break	$5.95	2	https://www.digikey.com/en/products/detail/ada
4 Pin Right Angle header	$0.49	1	https://www.digikey.com/en/products/detail/sulli
4A Switch	$3.56	1	https://www.digikey.com/en/products/detail/c-k/
Button	$0.10	1	https://www.digikey.com/en/products/detail/sam
Potentiometers	$3.22	2	https://www.digikey.com/en/products/detail/bou
2 Pin header	$0.26	3	https://www.digikey.com/en/products/detail/sulli
3 Pin Header	$0.33	2	https://www.digikey.com/en/products/detail/sulli
5 pin header	$0.42	1	https://www.digikey.com/en/products/detail/sulli
20 Pin Header	$1.10	2	https://www.digikey.com/en/products/detail/sulli
IC Sockets	$0.59	4	https://www.digikey.com/en/products/detail/prec
LEDs	$0.15	10	https://www.digikey.com/en/products/detail/w%0
Diode	$0.20	1	https://www.digikey.com/en/products/detail/stm
Inductor 100uH	$0.36	1	https://www.digikey.com/en/products/detail/bou
Pi Pico 2	$5.00	1	https://www.digikey.com/en/products/detail/rasp
20 Pin header male	$0.35	2	https://www.digikey.com/en/products/detail/ada
DAC - LTC1661	$7.16	1	https://www.digikey.com/en/products/detail/anal
LM386	$0.93	1	https://www.digikey.com/en/products/detail/texa
LCD w Backpack	$8.95	1	https://www.digikey.com/en/products/detail/sunf
1k Resistors	$0.10	11	https://www.digikey.com/en/products/detail/stac
10 Ohm Resistor	$0.43	1	https://www.digikey.com/en/products/detail/yage
3k resistor	$0.10	1	https://www.digikey.com/en/products/detail/stac
0.1uF Cap	$0.22	2	https://www.digikey.com/en/products/detail/kem
0.05uF Cap	$1.42	1	https://www.digikey.com/en/products/detail/vish
10uF Cap	$0.28	2	https://www.digikey.com/en/products/detail/ruby
100uF Cap	$0.35	1	https://www.digikey.com/en/products/detail/w%0
1uF Cap	$0.40	1	https://www.digikey.com/en/products/detail/ruby
1000uF Cap	$0.63	1	https://www.digikey.com/en/products/detail/ruby
Total	$51.86		

Figure 15: Bill of Materials with links. Google Drive Link

As seen, this does not include the cost of the PCB. Depending on the shipping, it can range from $40-$100 additional cost. This price compared to commercially available scoreboards is not that impressive. The original scoreboard for the game costs $140 at Game Room Guys (Game Room Guys). My scoreboard unfortunately does not include any mechanical parts to protect the circuit or make it look more like a scoreboard. However, mine does use an LCD which is more expensive than the seven segment displays they have. Overall, if your main goal is to get the bubble hockey scoreboard working, it may be more cost and time effective to purchase it directly.

Results

LCD Output

Figure 16: Select Screen Outputs

The above images are from the SELECT screen state. These are all toggled through using the potentiometer next to the Pico. The images on the right show example output when the game is running. If it is a game to a set time, the top shows the remaining time, and the score is on the left and right. For a game with a set score, then the time shows how long you have been playing, and the goals also reflect on the LEDs on the PCB.

Figure 17: Game Running Output

When someone scores, the left screen is shown for three seconds. After someone wins, the right screen is shown for 5 seconds, then is changed back to the original state. After the game is won, the LEDs are reset back to off, and a new game can be run.

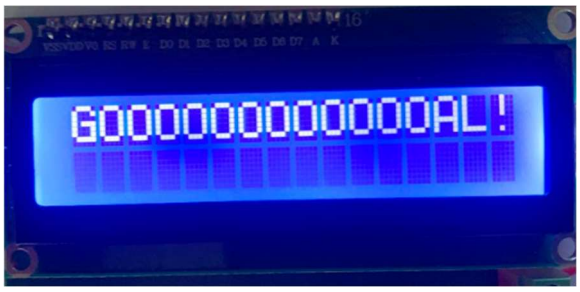

Figure 18: Game Ending and Score Screens

PCB

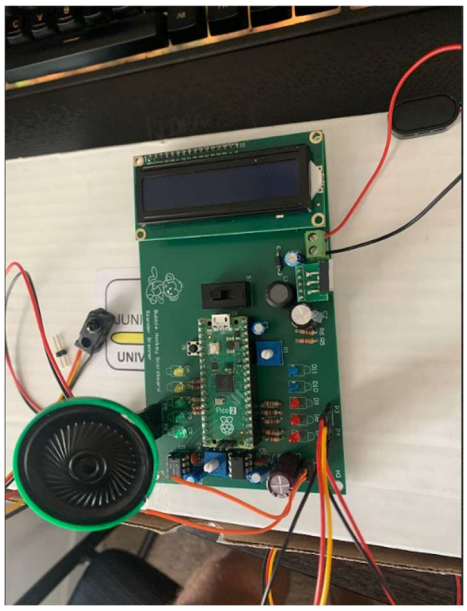

Figure 19: Completed PCB with Speaker and Externals

The above image shows the completed PCB. As you can see, the LCD takes up the top space of the PCB just above the power input. The beam break sensors plug into either side of the PCB corresponding to the LEDs and side they control. The On/Off switch is right above the Pico, but it unfortunately blocks the USB port. The Linear Regulator is in a separate PCB with a 5-pin header attached to it, so it plugs into the top 5 pin header.

Analog Output

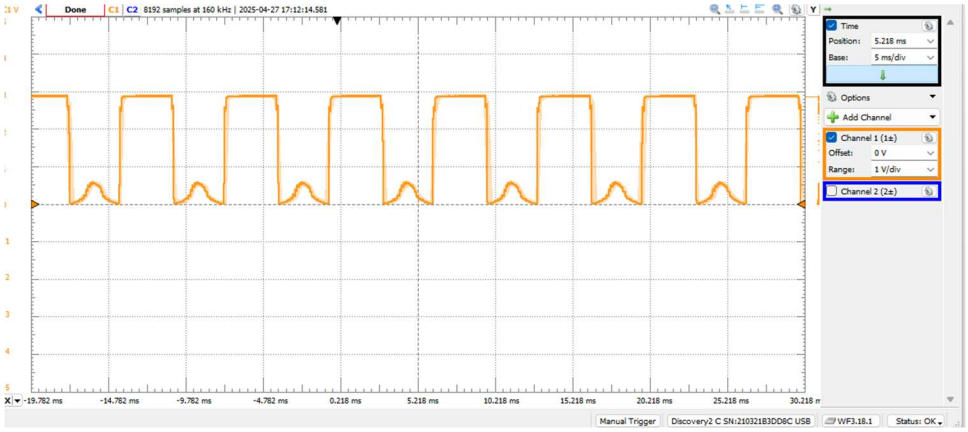

Figure 20: Output to Speaker from Audio Amplifier

The above image is the output to the speaker from the Audio Amplifier. As you can see, it is a square wave and not a sinusoid. It is in the correct frequency, but the issue was that my output from the DAC was in reference to 5V. Therefore, I was outputting a 5V sine wave, then the audio amplifier was amplifying it by 20. This immediately hits the rails, making my sine wave square. It still sounds like a buzzer, but in future renditions, the DAC should take the future amplification into account.

Discussion/Conclusion

Through the process of design to fabrication, there were many points in which I want to improve my device. However, I believe that the overall functionality is quite good. The goals and game are tracked efficiently, the LCD output is very clear and doesn't have much flashing from rewriting constantly.

For starters, I want to take into consideration the mechanical aspects of my PCB more in the next iteration. The scoreboard will be enveloped in some sort of box to hide the electronics, and having a large rectangle makes it more difficult to make the scoreboard aesthetically pleasing. Furthermore, I like how the LCD takes up the top of the PCB and sits flush, however it wastes lots of space that I am paying for during manufacturing. Instead, I think the screen can be connected with wires to the PCB and will be mounted to the outside of the box. The same applies to the speaker and battery pack, though they are already like that now. Overall, I'd shrink my PCB and fit more into a smaller space because I want the scoreboard to be enclosed in another box.

Furthermore, as discussed in the previous page, my audio output was not as intended. I forgot to factor in the automatic 20x amplification from the LM386. Therefore, my DAC needed to output only 0-~180mV. However, outputting the audio is very CPU intensive for high resolutions. If I want to eventually output voice which needs high levels of resolution, I will need to switch to an external solution such as a dedicated memory for the sounds that can output to the DAC and LM386. The speaker will also be placed in an enclosure to boost its sound quality.

A blatant fix is to move the switch away from the USB port of the Pico. I did not realize how tall it was, and it completely blocks the Pico. That means any code changes require removing the Pico, which can be very annoying to deal with.

Even with these changes, I believe that my project was very educational and complete. This process gave me the confidence to expand on my project in future renditions. Learning how to create the Gerber files to send to a manufacturer empowers me to continue developing and making physical products.

References

Abdulwahab.hajar. (2019, December 26). *Opamp vs audio amplifier??*. All About Circuits. https://forum.allaboutcircuits.com/threads/opamp-vs-audio-amplifier.135094/

Digikey. (n.d.). *Beam Break Sensor 2168*. Digikey Components. https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/44/2168_Web.pdf

Game Room Guys. (n.d.). *Shelti electronic scoring unit*. Retrieved April 30, 2025, from https://www.gameroomguys.com/Shelti-Electronic-Scoring-Unit_2

GeeksforGeeks. (2021, September 27). *Potentiometer - definition, working principle, types*. https://www.geeksforgeeks.org/potentiometer-definition-working-principle-types/

LM2576. LM2576 data sheet, product information and support | TI.com. (n.d.). https://www.ti.com/product/LM2576?utm_source=google&utm_medium=cpc&utm_campaign=app-null-null-GPN_EN-cpc-pf-google-wwe&utm_content=LM2576&ds_k=LM2576&DCM=yes&gad_source=1&gclid=CjwKCAiA5Ka9BhB5EiwA1ZVtvBxbvWcDZQZOHWMMQyEX_muTVNC3oFmcZB6QIdtnDqVA9DLMvrkDghoCtoAQAvD_BwE&gclsrc=aw.ds

OpenAI. (2025, April 27). *Response from ChatGPT [Large language model]*. OpenAI. https://chat.openai.com/

RealPars. (n.d.). *Photoelectric sensor explained (with practical examples) - realpars*. RSS. https://www.realpars.com/blog/photoelectric-sensor#:~:text=For%20the%20Through%2DBeam%20sensor,the%20sensor%20will%20turn%20off.

Storr, W. (2024, August 9). *Operational amplifier basics - op-amp tutorial*. Basic Electronics Tutorials. https://www.electronics-tutorials.ws/opamp/opamp_1.html

Ti. (n.d.). https://www.ti.com/lit/ds/symlink/lm386.pdf

Appendix

```
<main.py>
# library imports…
import hardware
import game_logic
import time
import constants

#lcd_timer = Timer(3)
#lcd_timer.init(freq = 0.5, mode = Timer.PERIODIC, callback=lcd_update)

def main():
# Init hardware
    player_a_leds, player_b_leds = hardware.init_LEDs()
    start, beam_a, beam_b = hardware.init_sensors()
    spi, cs = hardware.init_DAC()
    lcd = hardware.init_LCD()
    adc = hardware.init_adc()

    # Init game state
    game = game_logic.GameState()

    lcd_update_time = time.ticks_ms()
    lcd_update = {'update' : True}
    while True:
        #get flags
        flags ={
        'f_start' : hardware.button_pressed,
        'f_goal_a' : hardware.goal_a_flag,
        'f_goal_b' : hardware.goal_b_flag
        }


        if(time.ticks_ms()-lcd_update_time > 500):
            lcd_update['update'] = True
            lcd_update_time = time.ticks_ms()

        if game.state == 'SELECT':
            game_logic.handle_select(game, adc, flags, lcd, lcd_update)
        elif game.state == 'GAME_RUNNING':
            game_logic.handle_game_running(game, player_a_leds, player_b_leds, flags, lcd,
lcd_update)

        elif game.state == 'SCORE_SCREEN':
            game_logic.handle_score_screen(game, cs, spi, lcd, lcd_update)

        elif game.state == 'GAME_END':
            game_logic.handle_game_end(game, cs, spi, lcd, lcd_update)
            # make a new game
            game = game_logic.GameState()

        hardware.clear_flags()      # clear flags on each loop

        time.sleep_ms(10)  # loop timing

main()
</main.py>
<game_logic.py>
# game_logic.py is like the datapath and handles all the logic for each state
import time
from machine import Pin, SPI, I2C, ADC, Timer
```

```
import constants

GAME_OPTIONS = [
    {'mode': 'SCORE', 'target_score': 3},
    {'mode': 'SCORE', 'target_score': 5},
    {'mode': 'TIMER', 'target_time': 180},
    {'mode': 'TIMER', 'target_time': 300},
    {'mode': 'TIMER', 'target_time': 420},
]

sine_timer = Timer()
table_index = 0

class GameState:
    def __init__(self):
        self.state = 'SELECT'
        self.mode = 'SCORE'         # or timer
        self.target_score = 5
        self.score_a = 0
        self.score_b = 0
        self.target_time = 300      # in seconds, so 5 mins
        self.goal_time = 0
        self.start_time = time.ticks_ms()

def handle_select(game, adc, flags, lcd, lcd_update):


    # read potentiometer for char and map to 0-4 for state select
    val = (adc.read_u16() * len(GAME_OPTIONS)) // constants.MAXU16

    selected = GAME_OPTIONS[val]
    game.mode = selected['mode']
    game.target_score = selected.get('target_score', 0)     # gets target score if there is one
    game.target_time = selected.get('target_time', 0)       # same here
    # wait for button press → move to 'GAME_RUNNING'

        #update lcd if needed

    if(lcd_update['update']):
        lcd_update['update'] = False
        lcd.clear()
        lcd.putstr("Gamemode: ")
        lcd.move_to(0,1)
        if game.target_score != 0:
            lcd.putstr(f"{game.mode} to {game.target_score}")
        else:
            lcd.putstr(f"{game.mode} for {game.target_time}s")
    if(flags['f_start']):
        game.state='GAME_RUNNING'
        lcd_update['update'] = True
        game.start_time = time.ticks_ms()
    pass

def handle_game_running(game, leds_a, leds_b, flags, lcd, lcd_update):
    # update lcd if needed
    if(lcd_update['update']):
        lcd_update['update'] = False
        lcd.clear()
        #place time
        lcd.move_to(0, 0)
        lcd.putstr("A  TIME: " + get_game_time(game) + "  B")
        #place scores
        lcd.move_to(0, 1)
        lcd.putstr("{}".format(game.score_a))
        lcd.move_to(15, 1)
```

```
        lcd.putstr("{}".format(game.score_b))

    # check beam break → update score
    # if goal: change to 'SCORE_SCREEN'
    if(flags['f_goal_a']):
        game.score_a += 1
        game.state = "SCORE_SCREEN"
    if(flags['f_goal_b']):
        game.score_b += 1
        game.state = "SCORE_SCREEN"
    # update LED arrays
    for i in range(game.score_a):
        leds_a[i].value(1)
    for i in range(game.score_b):
        leds_b[i].value(1)

    # if timer or score limit: change to 'GAME_END'
    if game.mode == 'SCORE':
        if game.score_a >= game.target_score or game.score_b >= game.target_score:
            game.state = 'GAME_END'

    elif game.mode == 'TIMER':
        elapsed = time.ticks_diff(time.ticks_ms(), game.start_time) // 1000
        if elapsed >= game.target_time:
            game.state = 'GAME_END'
    pass

def handle_score_screen(game, cs, spi, lcd, lcd_update):
    # wait 3 seconds → return to 'GAME_RUNNING'
    #set LCD to Score!
    lcd.clear()
    lcd.putstr(f"GOOOOOOOOOOOAL!")

    # start sine_timer
    print("Sine Started")
    start_sine_wave(cs, spi)
    #sit in while loop
    start = time.ticks_ms()

    while time.ticks_diff(time.ticks_ms(), start) < 3000:
        pass
    print("Sine Stopped")
    stop_sine_wave()
    game.state = 'GAME_RUNNING'
    lcd_update['update'] = True
    pass

def handle_game_end(game, cs, spi, lcd, lcd_update):
    # wait 5 seconds → reset game → 'SELECT'
    # set LCD to winner of game
    lcd.clear()
    if(game.score_a > game.score_b):
        lcd.putstr(f"Team Red Wins!")
    else:
        lcd.putstr(f"Team Black Wins!")
    # start sine_timer
    start_sine_wave(cs, spi)
    #sit in while loop
    start = time.ticks_ms()

    while time.ticks_diff(time.ticks_ms(), start) < 5000:
        pass

    stop_sine_wave()

```

```
    game.state = 'SELECT' # redundant since itll be reset in main but here anyways
    lcd_update['update'] = True
    pass

def get_game_time(game):
    elapsed = time.ticks_diff(time.ticks_ms(), game.start_time) // 1000  # in seconds

    if game.mode == 'TIMER':
        remaining = max(0, game.target_time - elapsed)
        minutes = remaining // 60
        seconds = remaining % 60
        return f"{minutes:01}:{seconds:02}"  # e.g. "2:03"
    else:
        # Just show elapsed time in SCORE mode
        minutes = elapsed // 60
        seconds = elapsed % 60
        return f"{minutes:01}:{seconds:02}"


# FOR TIMER STUFF
# SPI Send Function
def send_to_dac(value, cs, spi):
    global table_index
    cs.value(0)
    buf=bytearray([constants.loadwakeA | (value >> 6), (value<<2) & 0xFC])
    spi.write(buf)
    cs.value(1)

# Timer Interrupt to Send Sine Wave
def sine_wave_callback(cs, spi):
    global table_index
    send_to_dac(constants.sineLUT[table_index], cs, spi)
    table_index = (table_index + 1) % len(constants.sineLUT)  # Loop through sine wave table

# Timer to generate 350 Hz sine wave
def start_sine_wave(cs, spi):
    sine_timer.init(freq=constants.freq_out*len(constants.sineLUT), mode=Timer.PERIODIC,
callback=sine_wave_callback(cs, spi))

# Stop the sine wave
def stop_sine_wave():
    sine_timer.deinit()
```

```
</game_logic.py>
<hardware.py>
```

```
from machine import Pin, SPI, I2C, ADC, Timer
from lcd_api import LcdApi
from pico_i2c_lcd import I2cLcd
import constants

# These will be updated by the ISR
button_pressed = False
goal_a_flag = False
goal_b_flag = False

button_timer = Timer()
goal_timer = Timer()

def init_LCD():
    #initialize your i2c module and lcd
    i2c = I2C(0, sda=Pin(constants.SDA_pin), scl=Pin(constants.SCL_pin), freq=400000)
    lcd = I2cLcd(i2c, constants.I2C_ADDR, constants.I2C_NUM_ROWS, constants.I2C_NUM_COLS)
    return lcd

def init_DAC():
```

```
253.
254.     #set up SPI to DAC
255.     sck= Pin(constants.pinSCK)
256.     mosi = Pin(constants.pinTX)
257.     spi = SPI(1, baudrate = 400000, polarity = 0, phase = 0, firstbit=SPI.MSB, sck=sck,
mosi=mosi, miso=None)
258.     cs = Pin(constants.pinCS,mode = Pin.OUT, value = 1)
259.     return spi, cs
260.
261. def init_adc():
262.     adc = ADC(Pin(constants.pinADC))
263.     return adc
264.
265. def init_sensors():
266.     # Buttons & goal sensor
267.     start_button = Pin(constants.pinButton, Pin.IN, Pin.PULL_UP)  # Start game
268.     beam_a = Pin(constants.pinSensorA, Pin.IN, Pin.PULL_UP)   # Goal detection
269.     beam_b = Pin(constants.pinSensorB, Pin.IN, Pin.PULL_UP)
270.
271.     start_button.irq(trigger=Pin.IRQ_FALLING, handler=button_handler)
272.     beam_a.irq(trigger=Pin.IRQ_FALLING, handler=beam_a_handler)
273.     beam_b.irq(trigger=Pin.IRQ_FALLING, handler=beam_b_handler)
274.     return start_button, beam_a, beam_b
275.
276. def init_LEDs():
277.     # 2 arrays of 5 leds each from constants.py
278.
279.     # simple for loop to get all pins
280.     player_a_leds = [Pin(pin, Pin.OUT) for pin in constants.player_a_pins]
281.     player_b_leds = [Pin(pin, Pin.OUT) for pin in constants.player_b_pins]
282.
283.     return player_a_leds, player_b_leds
284.
285. # ISRs that update globals to use in other files
286. def button_handler(pin):
287.     button_timer.init(mode=Timer.ONE_SHOT, period=50, callback=button_debounced)
288.
289.
290. def button_debounced(t):
291.     global button_pressed
292.     button_pressed = True
293.     button_timer.deinit()
294.
295.
296. def beam_a_handler(pin):
297.     goal_timer.init(mode=Timer.ONE_SHOT, period=2, callback=beam_a_debounced)
298.
299. def beam_a_debounced(t):
300.     goal_timer.deinit()
301.     global goal_a_flag
302.     goal_a_flag = True
303.
304. def beam_b_handler(pin):
305.     goal_timer.init(mode=Timer.ONE_SHOT, period=2, callback=beam_b_debounced)
306.
307. def beam_b_debounced(t):
308.     goal_timer.deinit()
309.     global goal_b_flag
310.     goal_b_flag = True
311.
312. def clear_flags():
313.     global button_pressed, goal_a_flag, goal_b_flag
314.     button_pressed = False
315.     goal_a_flag = False
```

```
316.     goal_b_flag = False
```

```
317. </hardware.py>

318. <constants.py>
```

```
319. # This file is for organization purposes
320.
321. ################ HARDWARE ########################
322. #globals
323. MAXU16 = 65535
324.
325. #I2C
326. I2C_ADDR     = 0x27
327. I2C_NUM_ROWS = 2
328. I2C_NUM_COLS = 16
329. SDA_pin = 0
330. SCL_pin = 1
331.
332. #DAC
333. pinSCK = 14
334. pinCS  = 13
335. pinTX  = 15
336.
337. #ADC
338. pinADC = 28
339.
340. #Buttons and sensors
341. pinButton = 2
342. pinSensorA = 3
343. pinSensorB = 4
344.
345. #LEDs
346. player_a_pins = [12, 11, 10, 9, 8]
347. player_b_pins = [16, 17, 18, 19, 20]
348.
349. #Sine LUT
350. sineLUT =[0x200, 0x240, 0x27f, 0x2bc, 0x2f6, 0x32c, 0x35e, 0x38a,
351.             0x3af, 0x3ce, 0x3e6, 0x3f6, 0x3fe, 0x3fe, 0x3f6, 0x3e6,
352.             0x3ce, 0x3af, 0x38a, 0x35e, 0x32c, 0x2f6, 0x2bc, 0x27f,
353.             0x240, 0x200, 0x1bf, 0x180, 0x143, 0x109, 0xd3, 0xa1,
354.             0x75, 0x50, 0x31, 0x19, 0x9, 0x1, 0x1, 0x9,
355.             0x19, 0x31, 0x50, 0x75, 0xa1, 0xd3, 0x109, 0x143,
356.             0x180, 0x1bf]
357.
358. freq_out = 300
359. table_index = 0
360. loadwakeA   = 0b10010000
361. </constants.py>
```

Plus the LCD codes given in previous lab.