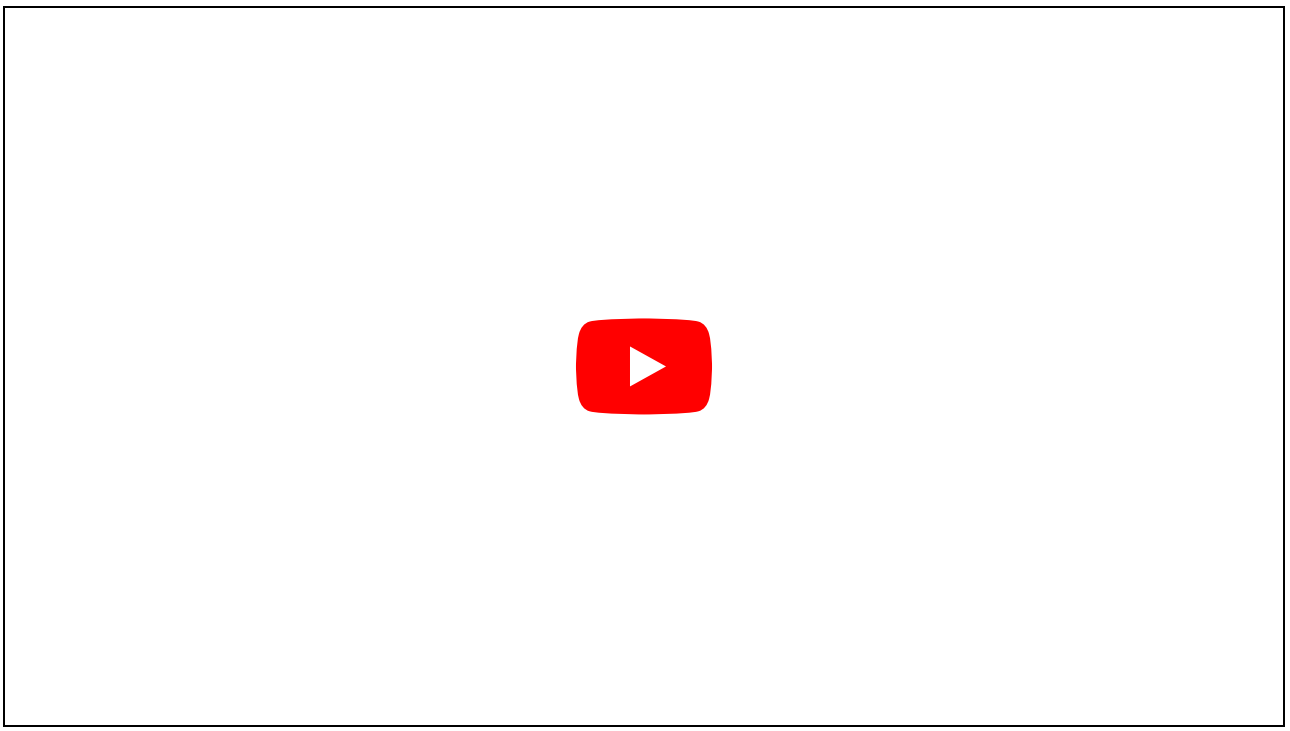
PiDeck

An All-in-One DJ Controller using Raspberry Pi

By Christopher Schiff and Justin Green



Demonstration Video

Introduction

At any club, you're likely to find a DJ using a high-tech controller to mix music. The technology behind mixing music has advanced significantly, allowing DJs to manipulate tracks in increasingly entertaining and innovative ways. However, commercial DJ controllers are often expensive and inaccessible, unnecessarily complicating entry into DJing as both a hobby and profession.

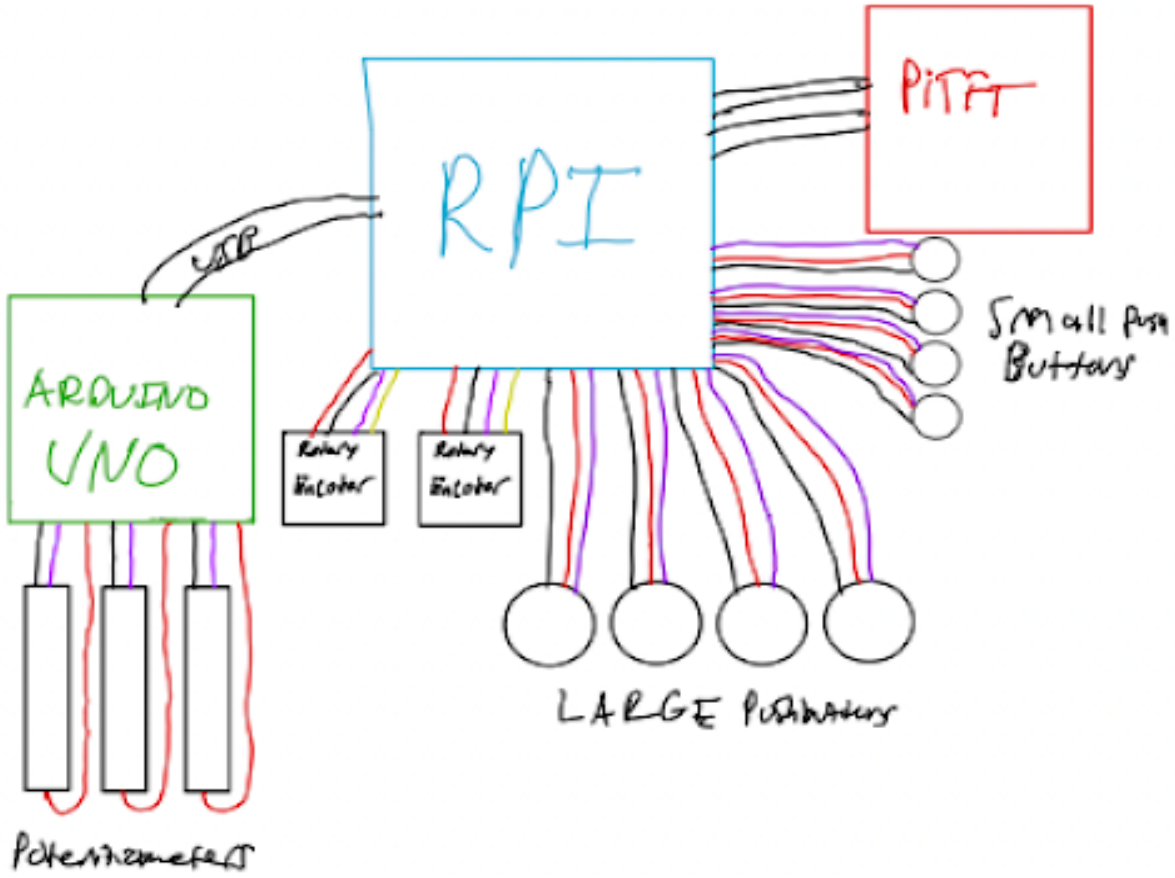
Commercial DJ Controller

Project Objective:

- Real-time playback and control of two audio tracks simultaneously
- Controls that are easy to understand and intuitive to use
- Visually appealing board
- Embedded design that doesn't require external software

Design

Hardware



Hardware Overview

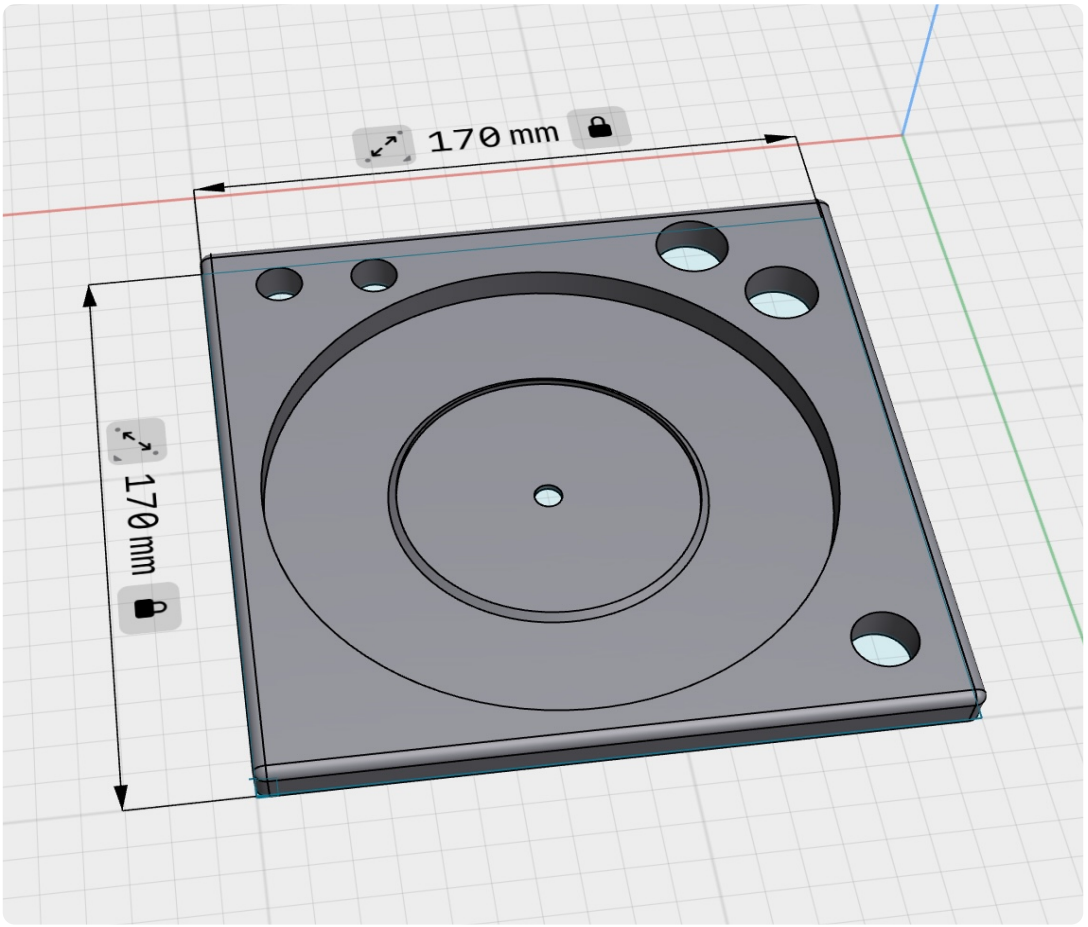
Our hardware setup for the Raspberry Pi consisted of 4 large push buttons, 4 small push buttons, 2 rotary encoders, and a piTFT display. Initially, we thought the hardware setup would be simple since the sensors chosen were not very complicated. However, we encountered a problem managing the sensors with the limited GPIO pins on the Raspberry Pi, especially since we needed to use the piTFT to display the song selection. We initially planned to use an ADC for our linear potentiometers since the Raspberry Pi does not have a built-in ADC. However, we ran out of room and decided to use an Arduino Uno, which has a built-in ADC. All three potentiometers were wired to the Uno, which was connected to the Pi via a USB connection to send the data from the potentiometers. Eventually, we ran out of space for two small push buttons, but they were still powered. Given the unique nature of the sensors chosen, please see the references for more information on the individual sensors. Here is the breakdown of the GPIO pins used, excluding the piTFT.

Sensor	**GPIO Pin #('s)**
Large Push Button	26
Large Push Button	19
Large Push Button	12
Large Push Button	16
Small Push Button	5

Sensor	GPIO Pin #('s)
Small Push Button	4
Rotary Encoder	20, 21
Rotary Encoder	13, 6

Figure 3: GPIO Pin Table

3D Printed Enclosure



The PiDeck DJ controller's enclosure was created using 3D printing for a custom fit for all hardware components. The design was made using Shapr3D CAD software, allowing precise parametric modeling to accommodate the Raspberry Pi, buttons, rotary encoders, and piTFT screen, with specific features for mounting.The final enclosure was 3D printed on a Bambu Lab A1 mini using PLA filament.

Software

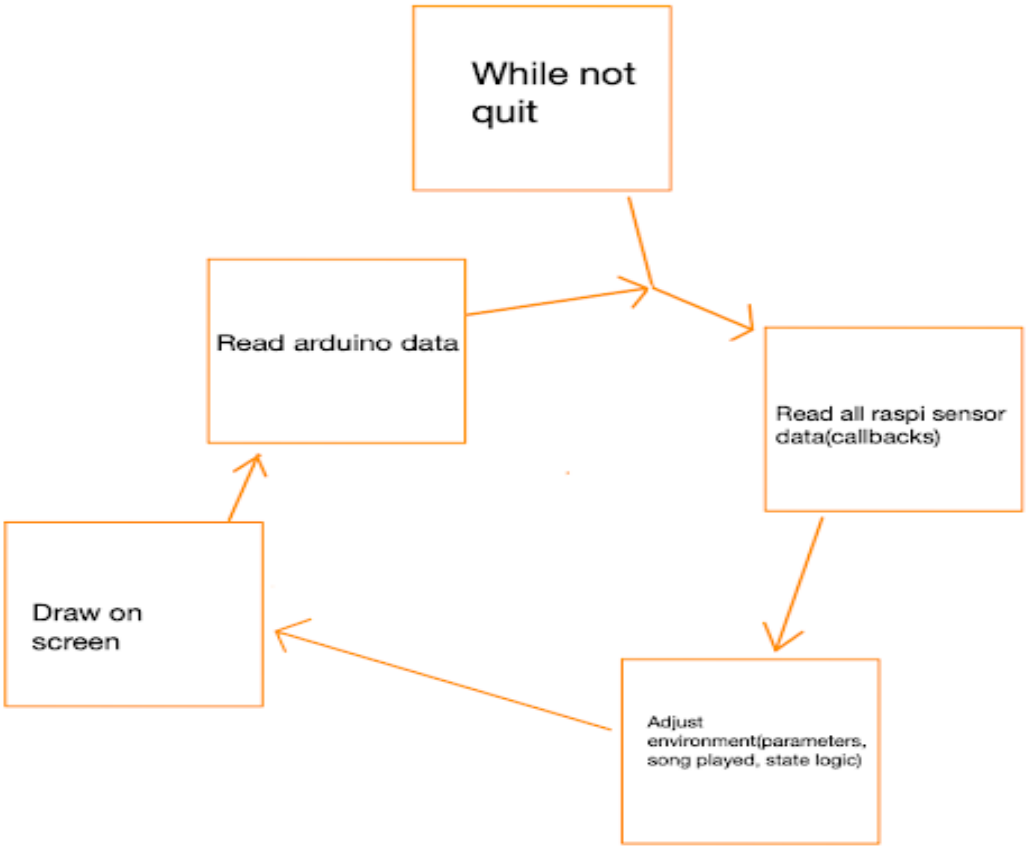
Pygame.Mixer

We ultimately had to make a big decision between pygame.mixer.Sound and pygame.mixer.music. The main trade-offs were the features we could implement. With Sound, we could add multiple channels, allowing us to have a dual song setup and add sound effects. With music, we could mutate a single song, including adjusting play speed and position, which would enable features like skipping through the song and changing the playback speed. We ultimately decided to go with

mixer.Sound because we believed the additional channels were the most valuable feature for our project.

Overall Logic Overview

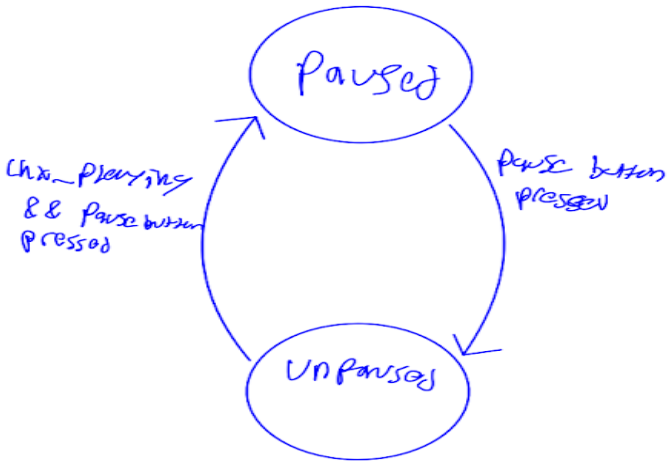
The RPi's software component of the final project contained one file, "final_pt3.py". In this file, we handle gathering all the data, playing music, and adjusting parameters. Below is an overview of the overall logic of the software.



Software Overview

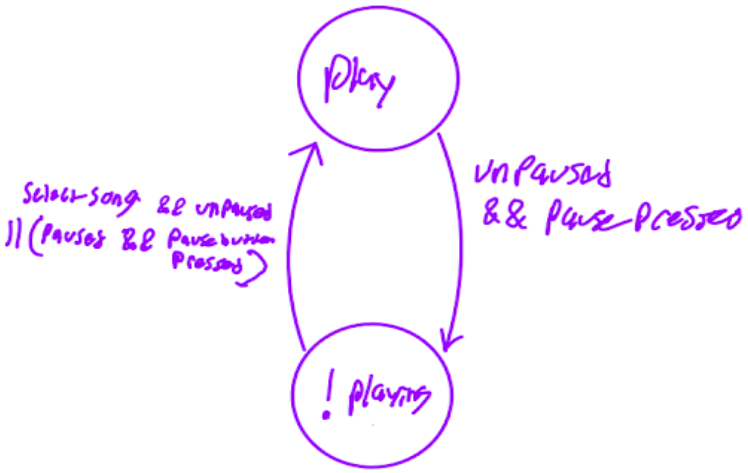
To walk through the logic, the program runs until a quit variable is set by pressing a piTFT button (GPIO 27). The program monitors all button presses and rotary encoder changes using callbacks. If an event occurs, the program executes the corresponding callback function, which requires adjustments in the environment. Firstly, the rotary encoders are used to scroll through songs, with their callbacks changing the index of the currently hovered song. Next, the potentiometer data, although not processed directly by the Raspberry Pi, are used to set the volume variables of the songs. The total volume of a song is composed of three variables: one for the associated potentiometer (left or right), one for the center potentiometer (crossfade), and one for an additional constant (overall scaling). Volume is contained in the range [0,1], with 0 meaning silent and 1 meaning maximum volume. The left potentiometer controls the volume of the song chosen on the left side of the screen, and the right potentiometer controls the volume of the song chosen on the right side of the screen. The crossfade potentiometer sets the volumes of the two songs relative to each other. If the slider is all the way to the left, the volume of the left song is maximized and the right is silent, and vice versa. Finally, an overall scaling constant can be increased by pressing the left small button and decreased by pressing the right small button. Depending on the speakers, the overall scaling of the volume may need to be adjusted to a higher level than others. The large push buttons control song selection and pausing. Each side has two buttons: the outer button is used for selecting songs, and the inner button is used to pause and unpause the song. To manage

the pause state, two interdependent state machines were created.



Paused State Machine Diagram

As demonstrated in the figure, the paused state (initially paused) will transition to unpaused when the pause button is pressed in that state. The other transition occurs when the other state machine is in the playing state and the pause button is pressed.



Playing State Machine Diagram

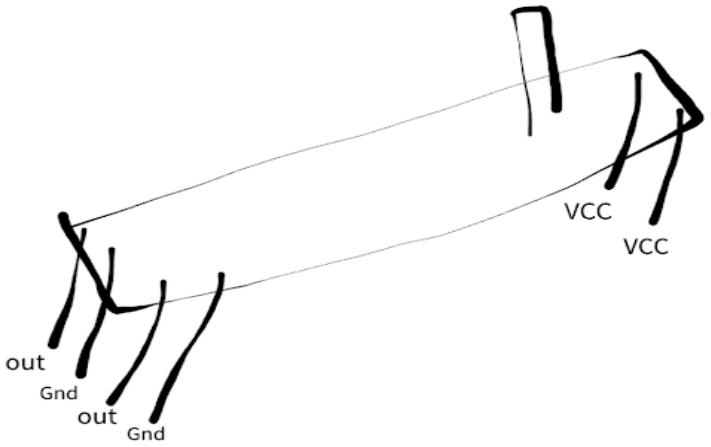
As demonstrated in the figure, the paused state (initially paused) will transition to unpaused when the pause button is pressed in that state. The other transition occurs when the other state machine is in the playing state, and the pause button is pressed. The other state machine controls whether a song is playing. If in the playing state, the song will transition to not playing if the paused state is unpaused and the pause button is pressed. The other transition was more difficult to design because drawing a progress bar for a song (discussed below) had issues with pausing. The two requirements are either a song is selected and unpaused (which only occurs before the first song is chosen) or the paused state machine is in the paused state and the pause button is pressed. An important note for both state machines is that since there were two songs played, there were two sets of each state machine implemented, one for each song. piTFT buttons were also incorporated. GPIO pins 17, 22, and 23 were used for sound effects, which would play on top of the selected music. These included a set DJ introduction, an airhorn sound, and a double scratch sound, although they are customizable. GPIO pin 27 is used to quit the program. Next, the screen is drawn with two main features. Firstly, all the song names are displayed in white, excluding the currently highlighted song (the song that the scroll is on), which is displayed in yellow. If the scroll changes, the highlighted song changes. Secondly, the selected song is displayed below along with a progress bar showing the percentage of the song that is completed. We found the duration of the song and, with every iteration, updated the bar to reflect the current time minus the start time and

paused time. From a coding perspective, this was the most difficult part to figure out because our paused state machine was initially limited. We had to tweak our transitions for the unpaused to paused state due to a few issues. The three issues we encountered were: when we switched songs, the progress bar wouldn't reset; when we paused and then unpaused, the bar would reset to blank; and when we paused then unpaused, the bar would jump to the percentage of (current time - initial start time) / duration. Finally, the data from the potentiometers are received from the Arduino (which just uses a filtered analog read).

The selected songs and sound effects are customizable but not during the program. To substitute songs, you must add .wav files to the audio_files directory contained in the project. If that is an issue, .mp3 and .mov files work if you run the bash scripts contained in the folder. For visibility purposes, we recommend a maximum of 8 songs. To customize the sound effects, .wav files must be added to the root of the project, and their names must be sound1.wav, sound2.wav, and sound3.wav, where the number corresponds to the sound effect from the top piTFT button to the third from the top.

Hardware Guide

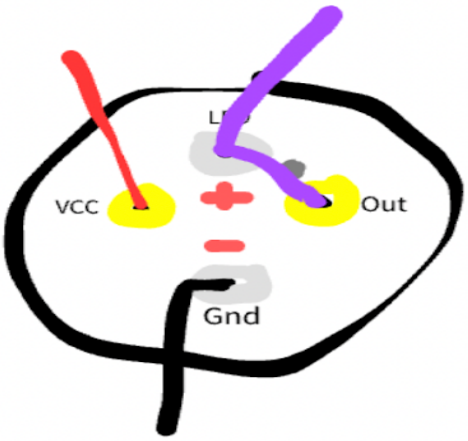
Slide Potentiometer



Slide Potentiometer Pinout

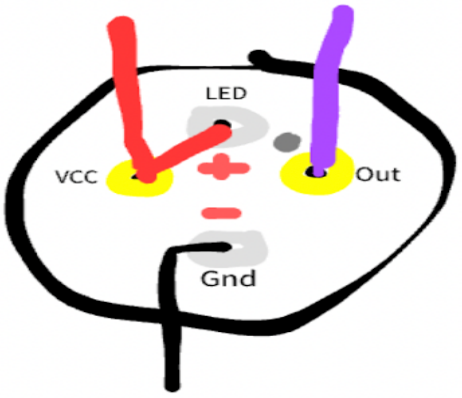
Although there is no data sheet given, through much debugging, we discovered a working model and integrated three into our final project. There are six pins contrary to the three given. Either pair of the left or right 3 pins(1 out, 1 Gnd, 1 Vcc). With a 10kohm built in the resistor, the sensor handles input voltage values of 3.3V to 5V although 5V is preferred. Through scoping the output values, the peak output of Vin occurs when the slider is all the way toward the side of the Vcc The potentiometer needs to use an ADC for useful values with the Raspberry Pi. If using Arduino, just plug the output into an analog pin due to the built in ADC, and call analogRead(pin#) to store the data. Otherwise, here is one example of a common ADC. Although advertised as linear the potentiometer is most definitely exponential. The max value depends on the sensor itself, but generally log10(max) ~ 2.95 with a 5V power. The logarithmic results showed to be very linear. To capture the smallest values of the potentiometer and to prevent errors, when calling log10(value) remember if value = 0.00 then set it to 1 so it won't cause a log(0) error.

LED Push Buttons



Push Button Pin Wiring (LED on when pushed)

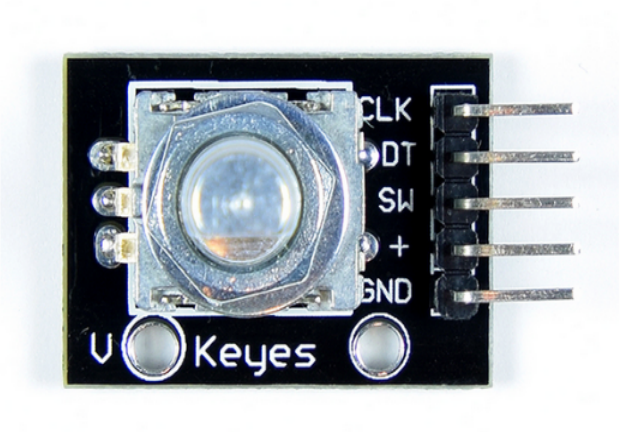
There are two options to wire up the push buttons depending on a choice of what you want to do with the LED. If you want to turn on the LED when the button is pressed and then turn off the LED when the button is pressed another time this is the wiring needed.



Push Button Pin Wiring (LED always on)

If you want to have the LED permanently on and the button to act as a peripheral this is the wiring needed. Luckily enough, the software was not difficult to establish. They are controlled exactly the same as piTFT buttons. One GPIO pin is needed and the setup is as follows. The pin is read as GPIO IN and is needed to be pulled down. (“GPIO.setup(Pin#, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)”). To read the data make sure to read on the rising edge for cleaner reads.

Rotary Encoders



Rotary Encoder

Wiring the Rotary Encoders The project uses two rotary encoders to control song selection and other functionalities. Each rotary encoder has two signal pins (A and B) and a common ground. For Encoder 1, pin A is connected to GPIO 20, pin B is connected to GPIO 21, and the ground is connected to a common ground on the Raspberry Pi. For Encoder 2, pin A is connected to GPIO 13, pin B is connected to GPIO 6, and the ground is connected to a common ground on the Raspberry Pi. It is crucial to connect the ground pins of the encoders to a common ground on the Raspberry Pi to ensure proper functioning. Using Rotary Encoders in PiDeck The rotary encoders are used to navigate through the song list and select tracks. The GPIO pins for the rotary encoders are set up as inputs with pull-up resistors, ensuring that the pins read high until they are actively pulled low by the encoder. The script includes functions to handle the rotary encoders' input by monitoring the state changes of pins A and B. By determining the sequence of these changes, the script can identify the direction of rotation (clockwise or counterclockwise). Based on the direction of the encoder rotation, the script updates the song list index, allowing users to scroll through available tracks. The song list index is incremented or decremented, and the display is updated to reflect the current selection. A separate thread continuously monitors the state of the rotary encoders. This thread ensures that any changes in the encoder's position are immediately detected and processed, providing real-time feedback to the user. This setup allows for intuitive navigation through the song list, enhancing the user experience of the PiDeck DJ controller. The rotary encoders provide a seamless way to select and manage tracks, making the controller more user-friendly and efficient.

Results

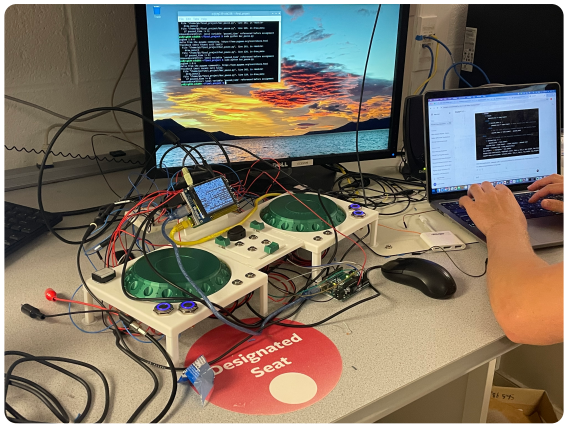
Conclusions

Ultimately, we put together a final project that encompassed all of our goals. Although we were not able to mutate music to speed up or slow down the songs, we incorporated several other features to manipulate music and provide entertaining experiences. Volume control, stop and start control, sound effects, and dual song playback allow for many options for customization. Our choice of pygame.mixer.Sound was crucial in allowing us to play multiple songs simultaneously and add interesting sound effects. Drawing to the piTFT was important to give the DJ a clear sense of the progress of the music and the options for choosing songs. The physical board is a reasonable size and can be easily controlled by a single user.

Future Work

If given more time, there are four changes we would make to our DJ controller to expand its functionality. Firstly, the design displayed on the piTFT had delays in the volume sliders, which need to be adjusted. Secondly, we would like to add more peripherals for greater control. Below is an example of the updated logic. Thirdly, in a future iteration of our DJ controller, we would hope to restructure our audio output to allow for changing the speed and position of the audio files. Finally, incorporating a method to upload songs live to the DJ board would provide more flexibility with music than the current limit of eight songs.

Work Distribution



Justin

jtg239@cornell.edu

- Sensors Debugger
- Music Selector
- Feature Planner
- Program Implementer
- Circuit Designer
- Early Riser

Christopher

cds258@cornell.edu

- 3D Designer
- Program Implementer
- Circuit Designer
- Late Sleeper

Parts List

- Raspberry Pi 4 - Provided in Lab
- Capacitive piTFT - Provided in Lab
- 2 x Rotary Encoder - Provided in Lab
- Arduino Uno with USB cord - $27.60
- 3 x 10K Ohm Slide Potentiometer (60 mm) - $13.49 for 5 (https://www.amazon.com/dp/B09D7P3RWZ?psc=1&ref=ppx_yo2ov_dt_b_product_details)
- 4 x 12mm Momentary Push Button w/ Green LED - $12.66 for 5 (https://www.amazon.com/dp/B0C38PBG9R?psc=1&ref=ppx_yo2ov_dt_b_product_details)
- 4 x 12mm Momentary Push Button w/ Blue LED - $12.66 for 5 (https://www.amazon.com/dp/B0C38PWG2B?psc=1&ref=ppx_yo2ov_dt_b_product_details)
- 4 x 19mm Momentary Push Button w/ Blue LED- $13.98 for 5 (https://www.amazon.com/dp/B08L13JG8M?ref=ppx_yo2ov_dt_b_product_details&th=1)
- 2 x 16mm Rectangular Push Button w/ White LED* - $9.99 for 5 (https://www.amazon.com/dp/B01MRWL6JS?psc=1&ref=ppx_yo2ov_dt_b_product_details)
- 1 x LED Rocker Switch* - $8.66 for 4 (https://www.amazon.com/dp/B07T5BFC99?psc=1&ref=ppx_yo2ov_dt_b_product_details)
- Resistors and Wires - Provided in lab
- *Included in physical enclosure but unused in project

References

Pygame (https://www.pygame.org/wiki/tutorials)
Pygame.Mixer (https://www.pygame.org/docs/ref/mixer.html)

UART Arduino Raspi (https://roboticsbackend.com/raspberry-pi-arduino-serial-communication/)
piTFT (https://learn.adafruit.com/adafruit-2-8-pitft-capacitive-touch/downloads)
R-Pi GPIO Document (https://sourceforge.net/p/raspberry-gpio-python/wiki/Home/)

Code Appendix

RPi Code (final_pt3.py)

```
import os
import pygame
import RPi.GPIO as GPIO
import time
import pigame
from pydub.utils import mediainfo
from pydub import AudioSegment
import serial
import threading

os.putenv('SDL_VIDEODRV', 'fbcon')
os.putenv('SDL_FBDEV', '/dev/fb0')
os.putenv('SDL_MOUSEDRV', 'dummy')
os.putenv('SDL_MOUSEDEV', '/dev/null')
os.putenv('DISPLAY', '')

# Initialize Pygame
pygame.init()
pitft = pigame.PiTft()
size = width, height = 320, 240
screen = pygame.display.set_mode(size)
pygame.display.set_caption('Dual Channel Music Player')
font = pygame.font.Font(None, 20)
font2 = pygame.font.Font(None, 15)
background_color = (30, 30, 30)
text_color = (255, 255, 255)
highlight_color = (255, 200, 0)
outline_color = (255, 255, 0)  # Yellow outline for visibility

#Initialize PauseL
pauseL = False
firstL = True
paused_timeL = 0.0
paused_initL = 0.0
elapsed_timeL = 0.0
#Initialize PauseR
pauseR = False
firstR = True
paused_timeR = 0.0
paused_initR = 0.0
elapsed_timeR = 0.0

# Encoder 1
pin_a1 = 20  # Encoder 1 pin A
pin_b1 = 21  # Encoder 1 pin B
# Encoder 2
pin_a2 = 13  # Encoder 2 pin A
pin_b2 = 6  # Encoder 2 pin B


# Initialize GPIO
GPIO.setmode(GPIO.BCM)
buttons = {'quit': 27, 'sound1': 17, 'sound2': 22, 'sound3': 23,
'A1' :pin_a1, 'B1': pin_b1, 'A2': pin_a2, 'B2': pin_b2 }
```

```
for button in buttons.values():
    GPIO.setup(button, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# Initialize the mixer
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
channels = [pygame.mixer.Channel(0), pygame.mixer.Channel(1), pygame.mixer.Channel(2)]
music_directory = 'audio_files'
songs = sorted([f for f in os.listdir(music_directory) if f.endswith('.wav')])
indexes = [0, 0]  # Indexes for each channel's current selection
currently_playing = [0, 0]
active_list = 0   # Initialize active_list

# Load durations using Pydub
def get_durations(songs, music_directory):
    durations = []
    for song in songs:
        song_path = os.path.join(music_directory, song)
        audio_info = mediainfo(song_path)
        duration = float(audio_info['duration'])
        durations.append(duration)
    return durations

song_durations = get_durations(songs, music_directory)
start_times = [None, None]  # Start times for each song
label_max = 20

def play_song(channel_index, song_index):
    track = pygame.mixer.Sound(os.path.join(music_directory, songs[song_index]))
    channels[channel_index].play(track)
    start_times[channel_index] = time.time()
    currently_playing[channel_index] = song_index

def draw_menu():
    global paused_timeL, firstL, elapsed_timeL, pauseL, paused_initL, paused_timeR, fi
rstR, elapsed_timeR, pauseR, paused_initR
    screen.fill(background_color)
    y_offset = 8
    for i, song in enumerate(songs):
        for j in range(2):  # Assuming two channels
            x_offset = 10 + j * 150
            is_active = indexes[j] == i #and j == active_list
            color = highlight_color if is_active else text_color
            song_label = f'> {song}' if is_active else song
            if len(song_label) > label_max:
                song_label = song_label[:label_max]
            text_surf = font.render(song_label, True, color)
            screen.blit(text_surf, (x_offset, y_offset + i * 20))

    # Draw progress bars with outlines and current song names
    bar_height = 20
    bar_width = 140
    for j in range(2):
        bar_x = 10 + j * 150
        bar_y = y_offset + len(songs) * 20 + 20  # Additional space for song name text
```

```
        if channels[j].get_busy() and start_times[j] is not None:
            # Display the current song name
            current_song_name = songs[currently_playing[j]]
            if len(current_song_name) > label_max:
                current_song_name = current_song_name[:label_max]
            song_name_surf = font2.render(current_song_name, True, highlight_color)
            screen.blit(song_name_surf, (bar_x, bar_y - 15))  # Position above the pro
gress bar

            # Calculate and draw the progress bar
            if j == 0:
                if pauseL:
                    if firstL:
                        paused_initL = time.time()
                        firstL = False
                    else:
                        paused_timeL = time.time() - paused_initL
                else:
                    if paused_timeL != 0:
                        firstL = True
                        start_times[j] += paused_timeL
                        paused_timeL, paused_initL = 0,0
                    elapsed_timeL = time.time() - start_times[j]
                # progress = elapsed_timeL / song_durations[j]
            else:
                if pauseR:
                    if firstR:
                        paused_initR = time.time()
                        firstR = False
                    else:
                        paused_timeR = time.time() - paused_initR
                else:
                    if paused_timeR != 0:
                        firstR = True
                        start_times[j] += paused_timeR
                        paused_timeR, paused_initR = 0,0
                    elapsed_timeR = time.time() - start_times[j]
                # progress = elapsed_timeR / song_durations[j]
            progress = elapsed_timeL / song_durations[j] if j == 0 else elapsed_timeR
/ song_durations[j]
            bar_length = int(progress * bar_width)
            pygame.draw.rect(screen, highlight_color, (bar_x, bar_y, bar_length, bar_h
eight))
            pygame.draw.rect(screen, outline_color, (bar_x, bar_y, bar_width, bar_heig
ht), 2)  # Outline

    pygame.display.flip()
    pitft.update()

def handle_quit(channel):
    pygame.quit()
    GPIO.cleanup()
    del(pitft)
    import sys
    sys.exit(0)
```

```
    exit()

def handle_select(channel):
    channels[active_list].stop()
    play_song(active_list, indexes[active_list])

def handle_scroll(channel):
    indexes[active_list] = (indexes[active_list] + 1) % len(songs)
    draw_menu()

def scroll_wheel(id, amt):
    #print(f"scroll {id},{amt}")
    global indexes, songs
    indexes[id] = (indexes[id] + amt) % len(songs)
    draw_menu()


def handle_switch(channel):
    global active_list
    active_list = 1 - active_list
    draw_menu()



#GPIO.setup([pin_a1, pin_b1, pin_a2, pin_b2], GPIO.IN, pull_up_down=GPIO.PUD_UP)

# Counters for each encoder
counter1 = 0
last_state1 = None
counter2 = 0
last_state2 = None

#define volume parameters
vol = 1



count0 = 0
count1 = 0

clk0LastState = GPIO.input(pin_a1)
clk1LastState = GPIO.input(pin_a2)

def handle_rotary0():
        global count0, clk0LastState
        clk0State = GPIO.input(pin_a1)
        dt0State = GPIO.input(pin_b1)
        if clk0State != clk0LastState:
                if dt0State != clk0State:
                        count0 += 1
                        scroll_wheel(0,-1)
                else:
                        count0 -= 1
                        scroll_wheel(0,1)
                #print(f"count0: {count0}")
```

```
        clk0LastState = clk0State

def handle_rotary1():
        global count1, clk1LastState
        clk1State = GPIO.input(pin_a2)
        dt1State = GPIO.input(pin_b2)
        if clk1State != clk1LastState:
                if dt1State != clk1State:
                        count1 += 1
                        scroll_wheel(1,1)
                else:
                        count1 -= 1
                        scroll_wheel(1,-1)
                #print(f"count1: {count1}")
        clk1LastState = clk1State


# Callback function for Encoder 1
def rotary_callback1():
    #print("rcb1")
    global counter1, last_state1
    state_a = GPIO.input(pin_a1)
    state_b = GPIO.input(pin_b1)
    if state_a and state_b:
        if last_state1 == "CW":
            counter1 -= 1
            scroll_wheel(0,-1)
        elif last_state1 == "CCW":
            counter1 += 1
            scroll_wheel(0,1)
        #print("Encoder 1 Counter: ", counter1)
    elif state_a and not state_b:
        last_state1 = "CW"
    elif not state_a and state_b:
        last_state1 = "CCW"

# Callback function for Encoder 2
def rotary_callback2():
    #print("rcb2")
    global counter2, last_state2
    state_a = GPIO.input(pin_a2)
    state_b = GPIO.input(pin_b2)
    if state_a and state_b:
        if last_state2 == "CW":
            counter2 += 1
            scroll_wheel(1,1)
        elif last_state2 == "CCW":
            counter2 -= 1
            scroll_wheel(1,-1)
       # print("Encoder 2 Counter: ", counter2)
    elif state_a and not state_b:
        last_state2 = "CW"
    elif not state_a and state_b:
        last_state2 = "CCW"
```

```
def sound_effect1(channel):
    # Stop any current playback on the new channel
    channels[2].stop()
    # Play the sound effect on the new channel
    sound_effect = pygame.mixer.Sound('/home/pi/final_project/sound1.wav')
    channels[2].play(sound_effect)

def sound_effect2(channel):
    # Stop any current playback on the new channel
    channels[2].stop()
    # Play the sound effect on the new channel
    sound_effect = pygame.mixer.Sound('/home/pi/final_project/sound2.wav')
    channels[2].play(sound_effect)

def sound_effect3(channel):
    # Stop any current playback on the new channel
    channels[2].stop()
    # Play the sound effect on the new channel
    sound_effect = pygame.mixer.Sound('/home/pi/final_project/sound3.wav')
    channels[2].play(sound_effect)


# Attach callbacks
GPIO.add_event_detect(buttons['quit'], GPIO.FALLING, callback=handle_quit, bouncetime=
300)
GPIO.add_event_detect(buttons['sound1'], GPIO.FALLING, callback=sound_effect1, bouncet
ime=300)
GPIO.add_event_detect(buttons['sound2'], GPIO.FALLING, callback=sound_effect2, bouncet
ime=300)
GPIO.add_event_detect(buttons['sound3'], GPIO.FALLING, callback=sound_effect3, bouncet
ime=300)
# Add event detection for both encoders
# GPIO.add_event_detect(pin_a1, GPIO.BOTH, callback=rotary_callback1, bouncetime=100)
# GPIO.add_event_detect(pin_b1, GPIO.BOTH, callback=rotary_callback1, bouncetime=100)
# GPIO.add_event_detect(pin_a2, GPIO.BOTH, callback=rotary_callback2, bouncetime=100)
# GPIO.add_event_detect(pin_b2, GPIO.BOTH, callback=rotary_callback2, bouncetime=100)


GPIO.setup(26, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(19, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(12, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(16, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

GPIO.setup(4, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
GPIO.setup(5, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)

ch1_pause = False
ch2_pause = False
ch1_play = False
ch2_play = False

def left_left_cb(channel):
    # CH1 Select
    #print("CH1 Select")
    global ch1_play, ch1_pause
```

```
    if not ch1_pause:
      channels[0].stop()
      play_song(0, indexes[0])
      ch1_play = True

def left_right_cb(channel):
    # CH1  Pause
    #print("CH1 Pause")
    global ch1_pause, pauseL, ch1_play
    if(ch1_pause):
        channels[0].unpause()
        ch1_pause = False
        ch1_play = True
        pauseL = False
    else:
        channels[0].pause()
        if (ch1_play):
          pauseL = True
          ch1_pause = True
          ch1_play = False

def right_left_cb(channel):
    # CH2 Pause
    #print("CH2 Pause")
    global ch2_pause, pauseR, ch2_play
    if(ch2_pause):
        channels[1].unpause()
        ch2_pause = False
        ch2_play = True
        pauseR = False
    else:
        channels[1].pause()
        if ch2_play:
          ch2_pause = True
          pauseR = True
          ch2_play = False

def right_right_cb(channel):
    # CH2 Select
    #print("CH2 Select")
    global ch2_play, ch2_pause
    if not ch2_pause:
      channels[1].stop()
      play_song(1, indexes[1])
      ch2_play = True

GPIO.add_event_detect(26, GPIO.RISING, callback=left_right_cb, bouncetime=200)
GPIO.add_event_detect(19, GPIO.RISING, callback=left_left_cb, bouncetime=200)
GPIO.add_event_detect(12, GPIO.RISING, callback=right_right_cb, bouncetime=200)
GPIO.add_event_detect(16, GPIO.RISING, callback=right_left_cb, bouncetime=200)

def inc_volume(channel):
  global vol
  vol *= 1.1
```

```
def dec_volume(channel):
  global vol
  vol /= 1.1


GPIO.add_event_detect(5, GPIO.RISING, callback=inc_volume, bouncetime=200)
GPIO.add_event_detect(4, GPIO.RISING, callback=dec_volume, bouncetime=200)

def map_value(x, original_max=29.35, target_max=1):
    return x / original_max

ser = serial.Serial('/dev/ttyACM0', 9600, timeout=1)
ser.reset_input_buffer()


data = {'pot#l' : 0.5, 'pot#r': 0.5, 'pot#c': 0.5}
def update_volumes():
    # Calculate the effective volume for each channel
    # Channel 1 (Left) fades in as pot#c goes from 0 to 1
    # Channel 0 (Right) fades out as pot#c goes from 0 to 1
    left_volume = vol * data['pot#l'] * data['pot#c']  # Increases as pot#c goes from
0 to 1
    right_volume = vol * data['pot#r'] * (1 - data['pot#c'])  # Decreases as pot#c goe
s from 0 to 1

    # Set the volumes
    channels[1].set_volume(left_volume)
    channels[0].set_volume(right_volume)




def rotary_encoder_thread():
    global counter1, last_state1, counter2, last_state2

    global count0, clk0LastState, count1, clk1LastState
    while True:
        handle_rotary0()
        handle_rotary1()
        # rotary_callback1()
        # rotary_callback2()
        #thread.sleep(0.01)


# Create and start the rotary encoder thread
rotary_thread = threading.Thread(target=rotary_encoder_thread)
rotary_thread.daemon = True
rotary_thread.start()


try:
    while True:


        draw_menu()
```

```
        if ser.in_waiting > 0:
            line = ser.readline().decode('utf-8', errors='ignore').rstrip()
            if line:
                try:
                    # Split the line by commas and filter out empty strings
                    entries = filter(None, line.split(','))
                    # Loop through each non-empty entry and parse it
                    for entry in entries:
                        # Split the entry into key and value parts
                        key, value = entry.split(':')
                        # Clean up any whitespace and convert value to float
                        key = key.strip()
                        value = float(value.strip())
                        data[key] = map_value(value)
                    # Now you can use the data dictionary for further processing
                  #  print(data)  # Example usage
                    update_volumes()
                except ValueError as e:
                    print(f"Error parsing line: {line}, Error: {e}")

       # thread.sleep(0.01)

except:
    print("Caught exception")
    GPIO.cleanup()
finally:
    GPIO.cleanup()
    del(pitft)
```

Arduino Uno Code (potentiometer.ino)

```
        #define pot1 A0
#define pot2 A1
#define pot3 A2

float val;
String pos;

void setup() {
  pinMode(A0, OUTPUT);
  pinMode(A1, OUTPUT);
  pinMode(A2, OUTPUT);
  Serial.begin(9600);

}

void loop() {
  //Prints the value of ADC
  for (int chan=0; chan<3; chan++) {
    if (chan == 0) {
      val = analogRead(pot1);
      pos = "c";
    }
    else if (chan == 1) {
      val = analogRead(pot2);
      pos = "l";
    }
    else {
      val = analogRead(pot3);
      pos = "r";
    }
    if (val == 0.00) {
      val = 1;
    }
    val = 10 * log10(val);
    Serial.print("pot#");
    Serial.print(pos);
    Serial.print(": ");
    Serial.print(val);
    Serial.print(",\n");
  }


  delay(10);
}
```