Digital Audio Effects Player

Alex Chitsazzadeh
Elmore Family School of ECE
Purdue University
Lafayette, USA
achitsaz@purdue.edu

Aditya Sood
Elmore Family School of ECE
Purdue University
Lafayette, USA
sood12@purdue.edu

Kerway Tsai
Elmore Family School of ECE
Purdue University
Lafayette, USA
tsai152@purdue.edu

Vinay Jagan
Elmore Family School of ECE
Purdue University
Lafayette, USA
vjagan@purdue.edu

Derek Xavier Rosales
Elmore Family School of ECE
Purdue University
Lafayette, USA
rosale22@purdue.edu

Zichen Zhu
Elmore Family School of ECE
Purdue University
Lafayette, USA
zhu1239@purdue.edu

Austin Carlton Kinkade
Elmore Family School of ECE
Purdue University
Lafayette, USA
akinkade@purdue.edu

***Abstract* - This paper provides a research summary for our embedded systems project. Our idea involves designing and implementing a digital audio player which facilitates audio playback with various kinds of digital filtering effects. The motivation for our idea lies in iterating upon the existing audio player solutions with varying applications for students, audio-enthusiasts, and embedded DSP hobbyists. We compare our idea with existing market products and describe how our sole focus is on keeping it cost effective. We then delve into the system architecture which entails both the hardware and software aspects implemented for this project. The paper concludes by evaluating the product metrics via a qualitative look at a survey as well as a quantitative look at the quality of the three filters implemented in the Digital Audio Effects Player: Bass Boost, Treble Boost, and Tremolo Effect.**

***Keywords* - Embedded System, STM32 microcontroller, Digital Signal Processor (DSP), Fast Fourier transform (FFT), Inverse Fast Fourier transform (IFFT), Tremolo filter, Matrix display, OLED interface**

I. Introduction of Research Project

The following paper contains a holistic review for the development and prototyping of a "Digital Audio Effects Player". The Digital Audio Effects Player was developed using an STM32F413ZH Nucleo Board microcontroller (MCU). It was selected due to its low-level controls and its onboard floating-point unit (FPU) that enables it to compute Fast Fourier Transforms (FFTs) efficiently. The project reads uncompressed data stored in a .WAV file format on a micro-SD card, applies audio processing via FFT/IFFTs and optional time-domain filtering before delivering the audio output to the external speaker. The user interacts with the product through 2 main avenues; the first is a set of potentiometers that control the various audio filtering parameters, and the second being an OLED screen controlled via buttons that hosts a user interface (UI) to execute processes such as track selection, play/pause functionality, and displaying details of the current track. The project also features an LED matrix display that visualizes the frequency components of the selected audio in real-time.

The implementation of the above features involved a variety of embedded systems techniques. Minimizing CPU workload was a driving design goal, and was accomplished using DMA transfers, leveraging existing HAL drivers and libraries, and using a circular buffer to store data for a seamless stream of uninterrupted audio. The audio processing is done using both digital filters and FFTs. These topics will be covered in depth in the following sections.

A. Motivation

In today's world it is not uncommon for the average person to own a smart device, typically a smartphone. Over the past couple decades, these smart devices have quickly evolved into the ultimate multi-purpose tool, offering a single platform to accomplish a variety of tasks such as calling, messaging, browsing the internet, and playing music, the latter of which will be the focus of our discussion. While it is generally convenient to have audio listening capabilities easily accessible through a smart device, there is a sub-category of consumers that may be dissatisfied with this approach. Individuals in this category may include:

1) Those that do not own a smart device due to high cost or other reasons,
2) Those that only desire audio playback functionality and do not care for the other suite of features a smart phone provides, and
3) Those that are more interested in learning about the implementation of audio players and would like a more hands-on, customizable solution that they have unfettered control of.

Dedicated audio players do exist, and they usually range from expensive high-fidelity audio systems to cheap MP3 players. However, devices with modifiable audio effects generally only exist in extremely high price ranges. Thus, the novelty of our solution is to build a device that not only is cheap but also catered to audio enthusiasts who aim to listen to their tracks with high-fidelity audio and custom filtering/effects. This project aims to create a product that the user can apply custom audio effects to existing files while remaining an affordable, scalable, and fun solution.

B. Potential Applications

Our product has several niche applications, particularly in education, music practice and embedded audio processing. Firstly, it offers a unique and interactive way for students to explore digital signal processing (DSP) concepts, making it a valuable educational tool. By modifying the knobs, the student can audibly understand the effect of various audio modulation techniques, equalization and filtering. Moreover, the visual spectrum analysis representation would show them real-time Fast Fourier Transform of a signal, enabling them to make a correlation between binning of frequencies in a sample signal. Additionally, if we can cut down on the cost significantly, it can serve as an affordable alternative to expensive DSP development boards, offering a simplified platform for hobbyists looking to experiment with real-time signal modulation.

Secondly, it is a great choice for musicians and audiophiles who want a portable tool for sound experimentation and practice. Our product will be able to generate complex audio playback making it ideal for learning various music pieces, while real-time equalizer and tremolo effects allow users to tweak sound characteristics for creative expression. The portable form-factor with a microSD card provides unrestricted potential making it great for entry-level audio player for young users. Additionally, the pure-C implementation of our product should enable it to support further firmware modifications that add new features, making it a flexible testbed for exploring real-time audio processing on STM32 microcontrollers.

C. Existing Solutions and Competitors

While numerous music players exist on the market today, most are catered towards audio enthusiasts or consumers looking for high-fidelity music playback. After some research, the following table notes a few notable competitors, each with their price, weight, and light description.

TABLE 1: EXISTING MUSIC PLAYER COMPETITORS

Device	Price	Weight	Description
Activo P1	~$400	155g	High-resolution MP3 player with dual-DAC and AMP circuit, Bluetooth 5.3, and 64GB storage. [1]
Fiio JM21	~$180	156g	Budget-friendly audio player with USB DAC functionality, 32GB storage, and Bluetooth 5.0. [2]
Astell & Kern SR35	~$700	184g	Premium Quad-DAC player with high-fidelity playback, 64GB storage, and 2-step gain mode. [3]
Fiio M23	~$650	~300g	High-resolution lossless music player with USB DAC, adjustable gain modes, and 64GB storage. [4]
Sony NW-A306	~$350	113g	Lightweight, WiFi-compatible player with 14-36 hours of battery life and USB DAC support. [5]

Many of these existing solutions focus on delivering high-quality sound, providing good storage and battery life, and supporting multiple file formats. Another notable feature of all these examples is that they are all designed like an all-in-one smartphone. Several of these music players include app stores and over-the-top customization options excessive of a simple music player, thus the expensive price tags. They are designed for serious music enthusiasts who want the best listening experience while maintaining the high-tech smartphone capabilities.

The “Digital Audio Effects Player” will be different because it gives users a budget-friendly option purely for audio playing with custom audio effects that can be applied by the user to existing song files. Unlike other music players, it also features an LED dot matrix display that shows real-time changes in the music’s frequency spectrum, making it a fun and engaging experience. While competitors focus on high-fidelity playback, this player will provide users with the ability to customize their sound through the Bass Boost, Treble Boost, and Tremolo Effect filters. By utilizing an STM32 microcontroller and micro-SD storage, this player will have reduced costs, allowing for a wider demographic of potential consumers.

II. SYSTEM ARCHITECTURE

A. Hardware Overview

Figure 1 illustrates the overall hardware system diagram for our project. For the user interaction, 32 × 64 RGB LED dots are scanned and refreshed with parallel GPIOE [13:0] ports + DMA2 Stream1 for real-time display of the FFT spectrum, while a 128 × 128 OLED assumes the character output of the menu and track information. Five potentiometers control the low-frequency (Bass) gain/cutoff, tremolo depth, and high-frequency (Treble) gain/cutoff, and are connected to ADC1 IN0-IN4 respectively, and the sampled ADC results are carried by DMA2 Stream0; four buttons are connected to GPIOD [4:7], which are used navigating the menus. Each button press triggers an interrupt, followed by a one-shot timer callback where the button press becomes resolved, and eventually saved to an internal variable that is checked by the system whenever it needs to.

The STM32F413ZH is set up to run the Cortex-M4 core at a frequency of 96 MHz with DSP extensions and was chosen because of its optimal balance between horsepower and cost: 320 kB SRAM allowed us to double-buffer 16-bit stereo samples and store many sets of 1024 audio samples in circular buffers. It also offers multiple DMA streams and a flexible clock tree, which are key to flicker-free audio. Meanwhile, since our prototype needs to use a lot of GPIO pinouts as inputs and outputs, most of them are used for connecting OLED, LED matrix SDIO Adaptor, boosts, filters, etc. So, the STM32F413ZH has strong economic and functional advantages compared to the ESP32 and other development boards.

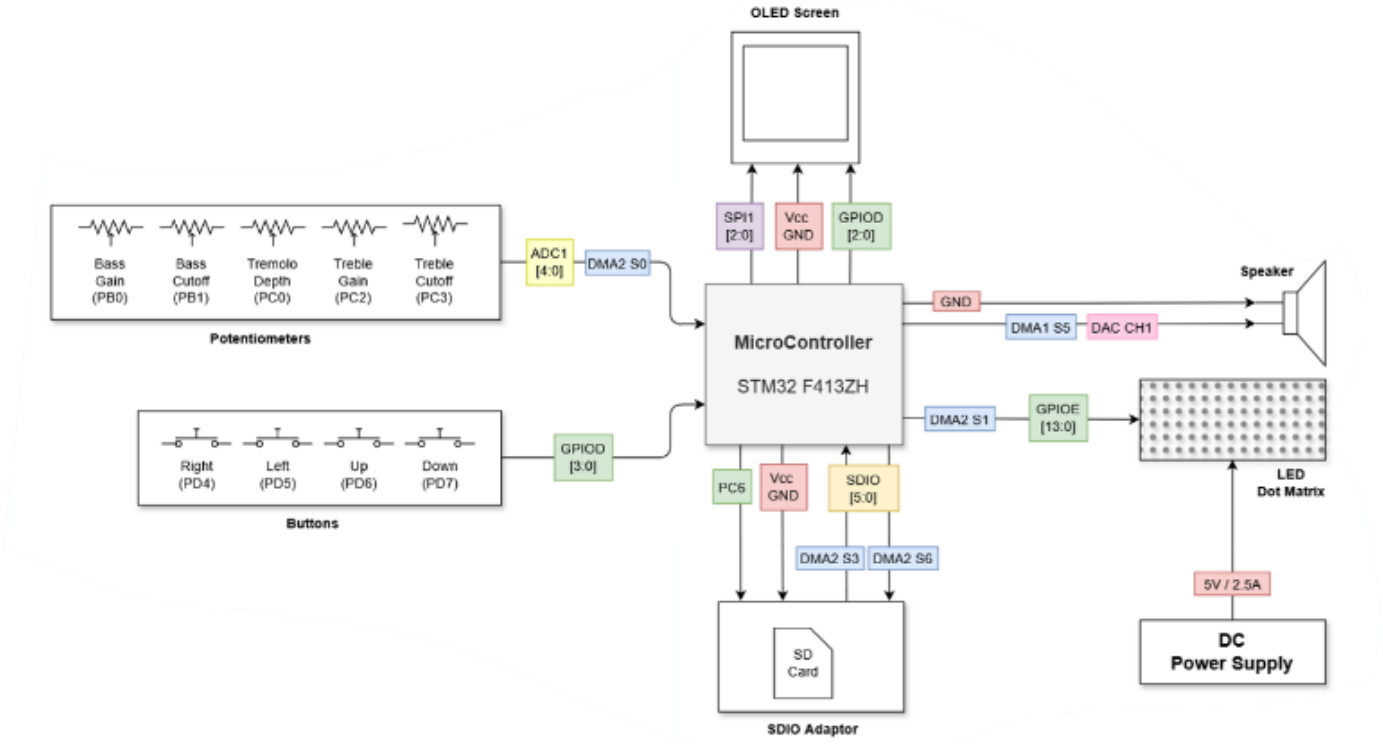

Figure 1: System Architecture Block Diagram

Several attempts were made to utilize the LM386 low-power audio amplifier IC to amplify and control volume from the output of DAC CH1 to the speaker. *Figure 2.* is the schematic of one such attempt.

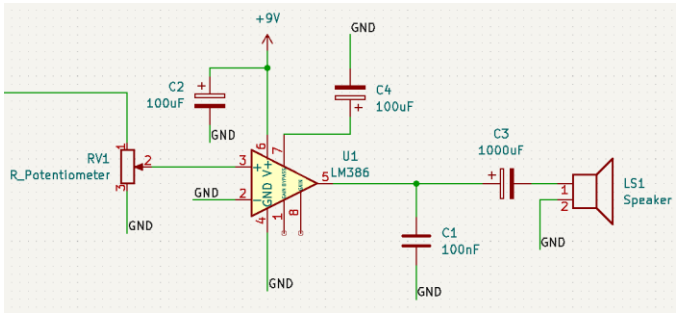

Figure 2: LM386 Amplifier Circuit

All attempts resulted in successful signal amplification and control, but the sound quality out of the DAC on the STM32 was unsatisfactory. While connecting the same circuits to an audio out line from an iPhone, the sound quality was satisfactory, however. This issue is likely due to some AC coupling issues coming out of the DAC on the STM32, likely causing distortion of the signal when entering the amplification circuit.

B. *Software Overview*

For the end-user to be able to easily load songs onto our product, we will need to use a file system to manage the data on our micro-SD card. Implementing a file system on its own can be a massive undertaking. Luckily, an existing open-source solution exists under the name of FatFs [6]. Utilizing such an API greatly simplifies work done on the data management front. On top of that, the need for a rather efficient implementation of the fast-fourier transform (FFT) arose relatively quickly in the planning phase of the project. The CMSIS-DSP software library included in our STM32CubeIDE development environment features an efficient implementation of the FFT that takes full advantage of the Arm Cortex M4F processor. Other than these two helpful libraries, the rest of the software has been written by us, from scratch, using the C programming language. *Figure 3.* below shows the overall software flow. We will be going over each part in-depth.

Due to the nature of the microcontroller, *Figure 3.* has been separated into both a flowchart for the CPU process, as well as the parallel actions conducted by the STM32F413 DMA engine. Initially, the MCU peripherals required for loading FatFs, communicating with the micro-SD card, and driving the OLED screen are initialized. This includes GPIO pins, certain DMA streams (the micro-SD card uses DMA2 Stream3 and DMA2 Stream6 for transferring data to and from the SDIO interface peripheral), and the SDIO, SPI, and ADC peripherals necessary for interfacing to the micro-SD card, OLED screen, and input potentiometers respectively. Once communication with the micro-SD card is confirmed via FatFs, the song directory is found, and a file scan is performed for all files in the .WAV format. The OLED drivers now kick in and render the initial "TRACK_LIST" screen, based on the state machine shown in *Figure 4.*

The state machine consists of four states. When in "TRACK_LIST_STATE", the OLED displays the track

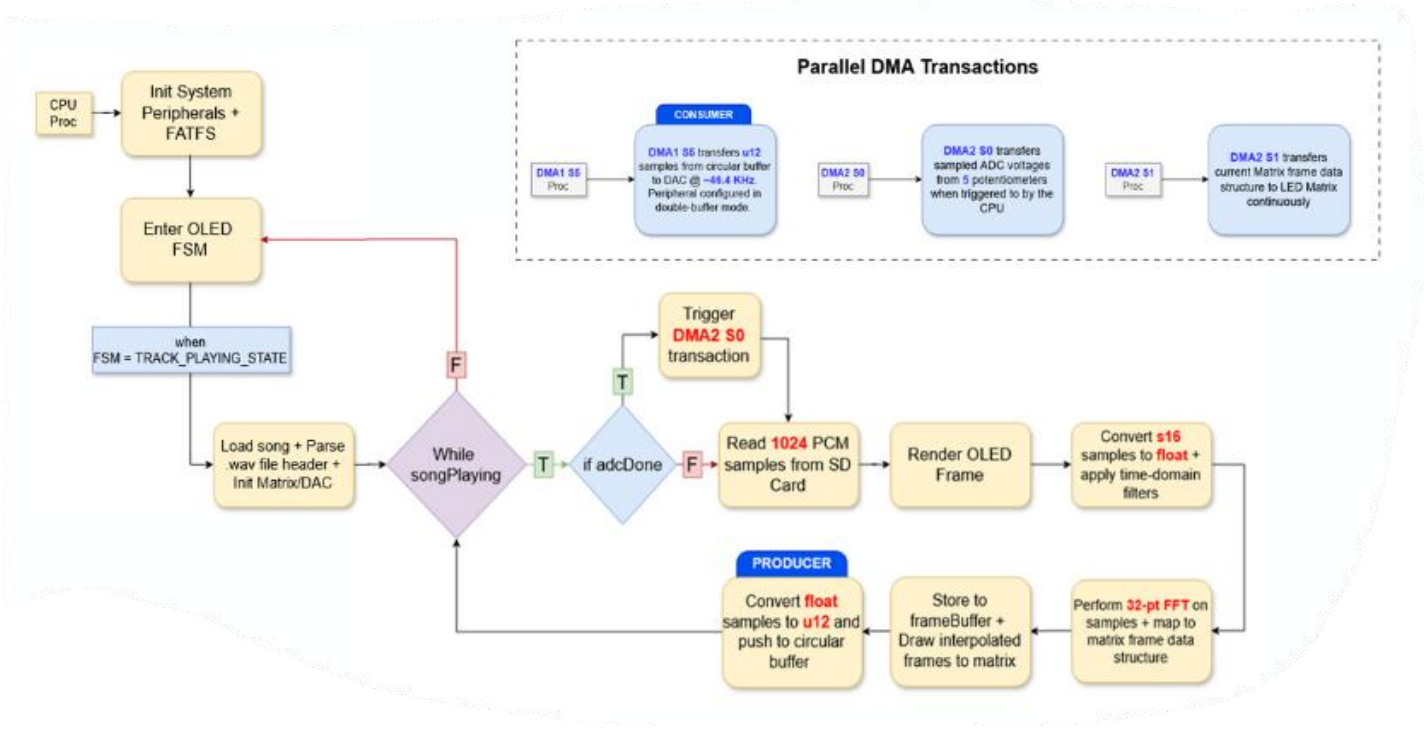

Figure 3: Software System Diagram

selection screen. Depending on user input, the OLED will then display an options screen in "OPTIONS_LIST_STATE".

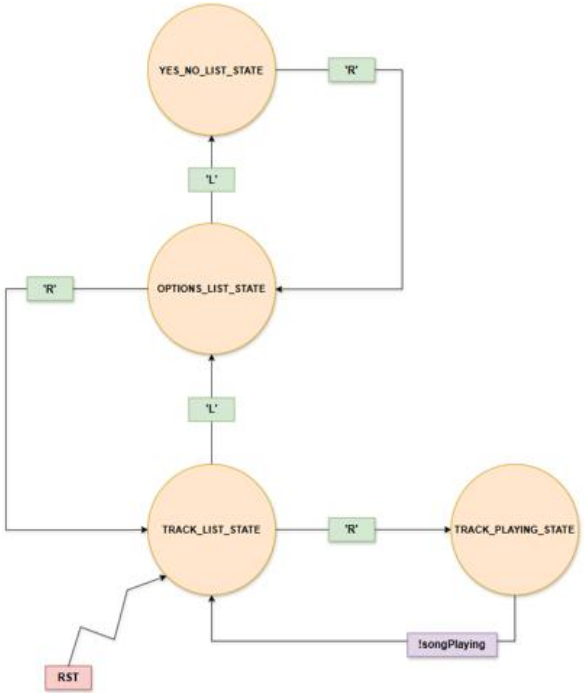

Figure 4: OLED Finite State Machine Transition Diagram

That screen can lead to a yes/no screen in the "YES_NO_LIST_STATE" where the setting selected by the user is recorded internally. The "TRACK_PLAYING_STATE" is special in that it starts playing the song, which involves a whole host of other operations. On top of playing the song, it draws a new frame on the OLED at periodic intervals to show how much time has elapsed in the song. The songPlaying flag is triggered when either the song is completed, or when the user wants to exit the song with the press of a button.

Figure 5: Track List OLED Screen

Figure 6: Options List OLED Screen

Figure 7: Yes/No List OLED Screen

Figure 8: Track Playing OLED Screen

The OLED screen is driven using the STM32's onboard SPI pins. The HAL drivers contain bindings that enable ease of access to the pins, as long as they are properly initialized either through code or the .ioc file. The OLED screen consists of 128x128 pixels that each utilize an 18-bit RGB space, where each set of 6 bits represents a color channel. Unlike a normal pin layout with just a clock and data pin, a separate pin exists that enables different subsets of operations: data or command. When the external pin is pulled low, the system can input a command to the OLED to control the screen. When pulled high, the system can input data into the screen's VRAM to either send continuous pixel data or large parameters.

A series of commands are initially sent to the OLED to configure the screen such as power on, resolution/contrast initialization, and VRAM addressing. To send pixel data, a command is sent to access VRAM and then the external pin is pulled high to send a large buffer of data to fill in VRAM. Once the 128x128 screen buffer is filled, the screen draws all at once.

Drawing on the screen works like a game loop when it comes to rendering. A large 32-bit image buffer of 128x128 entries exist where each element is a 32-bit value representing the color of the corresponding pixel. Since the RGB range is only 8 bits, only the lower 24 bits are considered. The 24 bits are split into 8 bits for R, G, and B and then normalized to a 6-bit value and then recombined to be sent into the screen's VRAM.

Draw functions were made to abstract a lot of the rendering capabilities. A draw_text function exists to draw text at a specified position. The function uses a C header file that contains an array bitmap of UTF-8 characters to be drawn. Other functions exist to draw different screens like the track list and the playing state.

Circling back to the flowchart in *Figure 4.*, once the "TRACK_PLAYING_STATE" is entered, the selected song is opened as a file on the micro-SD card. Since this project is limited to only reading .WAV files, we need to parse the .WAV header information which contains metadata such as how large the file is, the sampling rate of the song, the number of channels (mono/stereo), etc. The .WAV parser we developed performs byte swaps based on a comparison between the endianness of each chunk in the header and the endianness of the system it is running on. The .WAV parser was built to be as generic as possible, so that the code can be ported to systems other than the STM32F413ZH. Finally, the .WAV parser validates the header against possible values for each chunk, so that corrupted or improperly formatted .WAV files can be detected.

After the .WAV header has been parsed and validated, the LED matrix data structures are initialized. The LED matrix is 32 x 64 pixels and uses a proprietary protocol known as HUB75. Essentially, the matrix is constructed using a series of shift registers and latches, coupled with some strange quirks put into place by the creators to save pin real estate. Since the protocol must be "bitbanged" via GPIO on the STM32, we want to use DMA to waste as little time as possible. DMA2 Stream1 is set up to continually transfer data from the low-level current_matrix_frame data structure onto the GPIO pins connecting to the LED matrix display. One problem is that the LED matrix expects to be driven by a clock. To put it simply, HUB75 has 6 RGB pins that control the color of the pixel as well as 4 address pins that control the row. Two rows are shifted in at a time. Each time a set of RGB and address values are received, a clock is expected to synchronize the exchange of information. The simplest solution we could come up with was to double the size of the low-level current_matrix_frame data structure, where one half stores the RGB and address data with a low bit on the clock pin, while the other half stores the same RGB and address data but with a high bit on the clock pin. There is also a latch and blank pin that is handled similarly to the clock pin in the data structure. This way when the DMA is chipping away at the data structure and shifting in each pixel, the clock is automatically driven based on smart usage of the data structure. This also has the benefit of tightly coupling the clock with the arrival time of the incoming pixels.

The DAC data structures are also reset and initialized alongside the LED matrix data structures upon the start of a new song. The DAC peripheral is triggered at a frequency of ~46.4 KHz by a timer peripheral. Ideally, we would have liked to have a playback frequency of 44.1 KHz to match the sampling frequency of CD quality music, but ~46.4 KHz is the best we could do with how the base clock frequency was configured. If the CPU can keep up, then this means that the audio will sound slightly sped up due to the mismatch between the playback and sampling frequency. The timer is also triggering a DMA transaction on DMA1 Stream5. DMA1 Stream5 is configured in double-buffer mode, meaning that the buffer is divided into two separate segments. While segment A is being transferred to the DAC, segment B is updated. Once segment A is finished transferring, then the segments swap roles. Now segment B is being transferred to the DAC, while segment A is allowed to be written to. Overall, this helps improve the throughput of the system and ensures that the CPU can always update one of the DMA buffers, even if the DAC is not finished transferring from the other buffer. To be extra sure that the CPU does not run ahead and overwrite the buffer before the DMA has even had a chance to transmit either buffer to the DAC, we implemented a

circular buffer data structure. The CPU basically acts as a producer that pushes data to this circular buffer whenever it is ready, while the DMA acts as a consumer by popping the oldest set of samples from the buffer into its own free buffer upon completion of a DMA transfer via an interrupt handler. All these optimizations are necessary so that the CPU is never blocked by the DAC and can always move on to reading the next set of samples.

There is one more DMA engine (DMA2 Stream0) that is responsible for transferring the sampled ADC voltages from the potentiometers. There are five potentiometers, as shown in *Figure 1*., and they take a surprisingly long time to produce samples. Therefore, we will have the CPU check if the previous ADC transaction is completed. If it is, then the CPU issues a new one so that the latest values are updated. Otherwise, the CPU will move on. These sampled ADC values are used for controlling the gain, cutoff frequency, and depth for the filters.

The CPU's main responsibility is to fetch values from the micro-SD card by performing reads. The .WAV file data samples are encoded as signed 16-bit integers. 2048 bytes (1024 samples) are read out by the CPU every iteration (please see Section V. Limitations for further details on why this number was chosen). After that, the OLED frame is sometimes re-rendered. The reason why it is not always rendered is twofold. First, the OLED screen is 128x128, so to save CPU time we would have to make sure we only update the part of the screen that changed (that would be the progress bar and current elapsed time of the song in this case). Second, the OLED does not have a DMA engine attached to it, so it would waste even more CPU time to transmit all that screen data to the OLED via SPI. We could have implemented another DMA stream to take care of this, but we chose not to since there are already three higher priority tasks (ADC, Matrix display, and DAC DMA engines) competing for the memory bus alongside the CPU. Therefore, we only redraw the OLED screen at large time intervals, where the slight lag introduced during that iteration is not audible.

The 16-bit signed integers read from the micro-SD card are converted into floating point numbers to prepare the samples for processing. The packet is processed by a chain of three time-domain filters in the series (Bass Boost Filter, Treble Boost Filter, and Tremolo Filter). These filters can each be enabled or disabled in the "OPTIONS_LIST_STATE" screen of the OLED, as shown in *Figure 6.* We attempted to do this processing within the frequency domain by performing a 1024-pt FFT on the sample data, but that proved to be far too computationally expensive. Again, we are forced to read out 1024 samples from the micro-SD card each iteration due to the limitations of the system, so the frequency domain approach is not possible without introducing significant lag into the audio playback. The time domain approach proves to be simpler computation-wise, as it's just a series of first-order IIR filters that perform a few simple multiplications and additions over the 1024 samples.

Figures 9 and 10. depict our two complementary spectral-shaping filters - a bass boost and a treble boost - each built around a simple first-order IIR stage and a gain-and-sum topology. In *Figure 9.*, the input is first passed through a low-pass filter whose cutoff frequency is set by the user. The resulting "bassComponent" is then multiplied by a user-tunable boost factor and recombined with the dry signal, enriching the low end without overly coloring the midrange. *Figure 10.* follows the same pattern but flips the filter polarity: a high-pass stage isolates frequencies above the cutoff, the extracted treble band is amplified by a treble-boost coefficient, and this boosted high-frequency content is added back to the original waveform. By adjusting the cutoff and gain knobs, both filters allow smooth, musical enhancement of either end of the spectrum - bass for warmth and punch, treble for clarity and presence.

Figure 11. illustrates the tremolo effect, which belongs to the time-domain group of amplitude-modulation filters rather than a spectral boost. Here, the audio path is split into a "dry" branch (weighted by 1 – depth) and a "wet" branch (weighted by tremoloDepth), as shown in the two gain blocks feeding the summer. The wet branch is multiplied by a low-frequency triangle wave generated in the Modulator block (triFreq sets the LFO rate, triMax bounds the waveform). This triangle-wave LFO smoothly varies the amplitude of the wet branch, so when it's summed back with the dry path you hear a periodic swell and cut of volume—classic tremolo. By tuning the LFO frequency and depth parameters, *Figure 11's* architecture can produce everything from subtle ambience to driving, rhythmic pulsing.

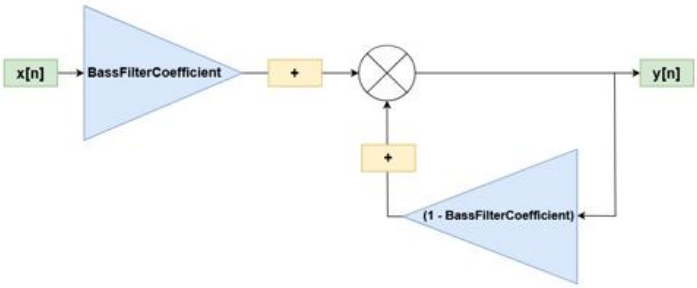

Figure 9: Bass Boost Filter Block Diagram

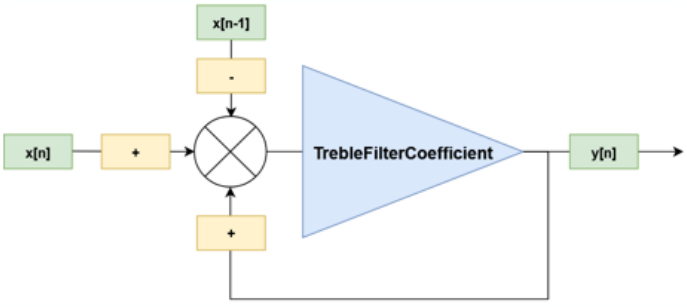

Figure 10: Treble Boost Filter Block Diagram

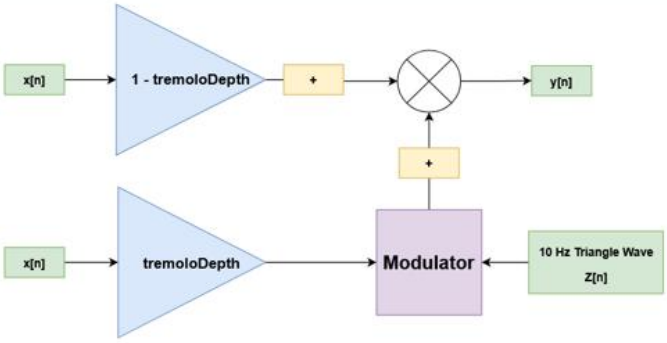

Figure 11: Tremolo Filter Block Diagram

Once the samples have been processed with the filters, we will want to take a 32-pt FFT of that data so we can draw the frequency spectrum of the live audio data on the LED matrix. A 32-pt FFT will produce 16 different frequency bins ranging from 0 to 22050 Hz, since the audio sampling rate is 44,100 Hz. 16 bins are appropriate for drawing onto the matrix since we have 64 columns of pixels to work with, leaving each bin to have a width of 4 pixels. The height of the bin is determined by first scaling the FFT output down by the maximum magnitude present in the FFT. Then the FFT magnitude can be directly correlated with bin height on the LED matrix by just viewing each magnitude as a percentage of the maximum height (32 pixels). Once the FFT is mapped onto the matrix, it is stored in a high-level current_matrix_frame data structure (not to be confused with the low-level current_matrix_frame data structure). This higher-level structure does not carry low-level information about the clock and latch pins that the matrix requires. It is simply a 32x64 grid that holds a color (black indicates no presence). This frame is pushed into a frameBuffer that can hold up to two frames. To ensure a smooth transition between any two FFT frames, we compute and display a series of interpolation frames in between any two frames. The interpolation is performed by performing a weighted average between the two frames in the frameBuffer over a series of steps. Each step varies how much weight is put on the current frame vs. the previous frame. We found that our system was limited to computing and drawing 4 interpolated frames in between every 2 key frames. Any more than that would introduce significant lag into the playback audio.

*Figure 12: LED Matrix Showing Frequency Spectrum of **Live Audio Playback***

Finally, we transform the processed floating-point samples into unsigned 12-bit integers that are pushed to the circular buffer that is later read by the DMA1 Stream5 engine that sending samples to the DAC. Because we are using a 12-bit DAC, there is going to be some aliasing in the audio quality (audio is recorded with 16-bit granularity), but it is not very noticeable. The CPU then restarts the loop with the next set of samples read from the micro-SD card. This process repeats itself until either the song ends, or the user terminates the song early with a push button press.

III. Evaluation

We used two methods for evaluating our Digital Audio Effects Player product. Method 1 involved collecting our audio output with and without the filters enabled and plotting graphs of the frequency spectrum to verify the quality of our filters. Method 2 is more of a quantitative method, where we put together a demo-video and then sent out a survey to people, asking for opinions on how our filter effects sounded.

A. Experimental Setup

For method 1, data was collected directly off the STM32F413ZH board by logging print statements via a serial terminal such as PuTTY and saving the session output to a .txt file. The STM32's UART interface proved to be handy here, as we were able to print out internal variables to a serial terminal on a host PC. We captured the DAC output samples for the first ~6 seconds of a song under the following four configurations:

- No filters enabled (x[n])
- Bass Boost Filter enabled w/ maximum gain and highest cutoff frequency (y[n])
- Treble Boost filter enabled w/ maximum gain and highest cutoff frequency (y[n])
- Tremolo Filter enabled w/ maximum tremolo depth (y[n])

For method 2, we captured a demo-video with 3 sample songs each applying a different audio effect. A google forms survey was then created including the video and the following two questions for each effect:

- On a scale of 1 (barely) to 5 (very), how noticeable was the change after applying a given effect?
- On a scale of 1 (poor) to 5 (good), how would you rate the quality of a certain effect?

The survey was shared with random people outside of our friend circle, as well as people within our friend circle, to get as much unbiased data as possible. Participants watched our demo video, heard the audio and rated each filter. In total, we recorded 24 observations for this method.

B. Data Collection Procedure

Once the various audio data for method 1 was recorded, we performed FFTs on the data using a Python script. To see the difference between the raw output with no filters enabled and the filter with the bass boost (or the filter with the treble boost),

we superimposed the frequency spectrums on the same graph. Since the tremolo filter is simply just an amplitude modulation scheme with a 10 Hz triangle wave as the carrier signal, we chose to do a time-domain comparison between the raw output signal with no filters and the output signal with the tremolo filter at max depth. Keep in mind that the Digital Audio Effects Player normalizes all the audio down in between each filter to prevent the audio from clipping due to excessive gain.

For method 2, we first obtained the rating data to investigate. We then created a clustered histogram to compare the relative results which are summarized in the next section. This helped us gauge an understanding of how our audio effects appeal to a wider audience.

C. *Results and Comparing with Existing Works*

Figures 13., 14., and 15. below showcase the difference between our audio signals before and after applying the filters. Another note is that the songs we are using don't have much frequency content above ~5 KHz, so we clipped the graphs to only show the [0, 5000] Hz range. For the bass boost filter, you can see that the lower frequencies in x[n] (orange) are amplified in the y[n] (blue) frequency spectrums. Since the cutoff frequency was relatively high here, as that is what the input potentiometer was set to when the experiment was conducted, the range of amplification was somewhat wider than just the extremely low frequencies. For the treble boost filter, the results look like they amplify some of the frequencies higher than the lowest frequencies amplified in *Figure 13.*, which means higher frequencies are being amplified, but then the rest of the frequencies in the song become greatly attenuated. This error could be caused because we are using a 1st order IIR filter for the treble boost, which has severe limitations in what it can or can't do. Finally, *Figure 15.* showcases the tremolo filter, which is best seen in time-domain since it is essentially an amplitude modulation scheme. Here, we are modulating the input with a low-frequency triangle wave (10 Hz). The time-slice from t=0.18 seconds to roughly t=0.205 seconds shows attenuation in the output signal y[n], possibly because at that time, the triangle wave signal is at the trough (bottom) of the wave. Later, during the t=0.24 to t=0.25 time-slice, the output signal is being amplified. In conclusion, this effectively shows the low-frequency oscillation of the audio signal x[n]'s peak amplitude over time.

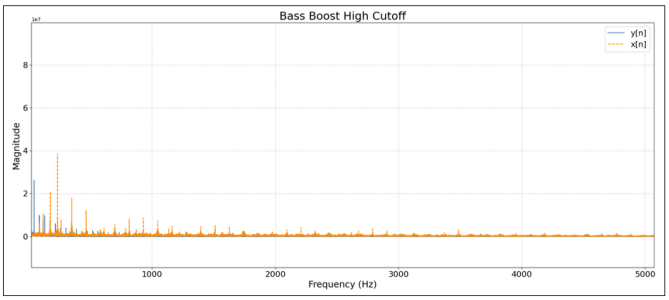

Figure 13: Bass Boost High Cutoff Frequency x[n] vs. y[n]

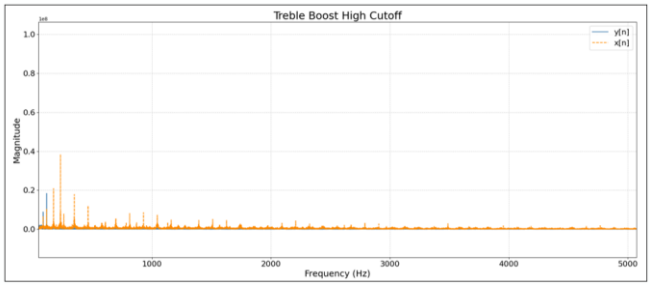

Figure 14: Treble Boost High Cutoff Frequency x[n] vs. y[n]

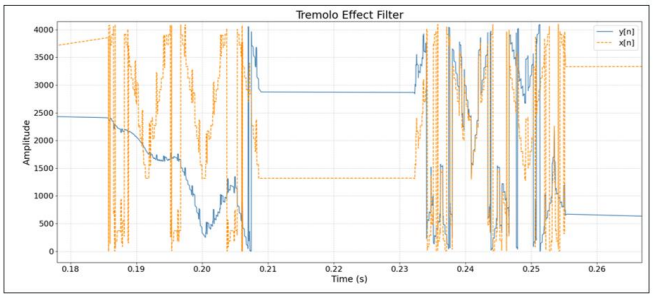

Figure 15: Tremolo Effect Filter with Max Tremolo Depth x[n] vs. y[n]

Figure 16. below shows how noticeable the users found three audio effects: Bass Boost, Treble Boost, and Tremolo Filter. Bass Boost had the most significant impact, with 12 people rating it a 5 ("Very" noticeable). Treble Boost and Tremolo Filter were generally rated as moderately to highly noticeable, peaking at ratings 4 and 5. Very few participants rated any effect as barely noticeable. Overall, Bass Boost stood out as the most perceptible effect among the three, with the Tremolo Filter not far behind it. The Treble Boost filter appeared to be a bit less noticeable according to the survey response.

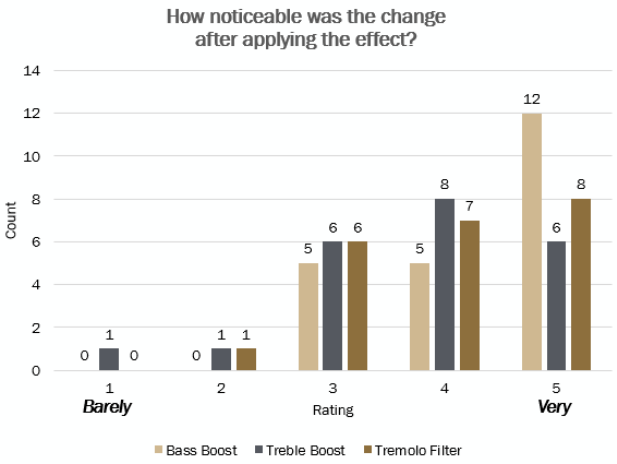

Figure 16: Survey Result for Question 1

Figure 17. illustrates how users rated the quality of the three effects. All three effects received the highest ratings at

4 and 5, with Treble Boost rated 4 by 11 people, which was the highest single count. Bass Boost also received strong positive feedback with 10 users rating it a 4. Lower ratings (1–2) were rare across all effects, suggesting generally favorable perceptions. Overall, Treble Boost slightly led in quality perception, followed closely by Bass Boost and then Tremolo Filter.

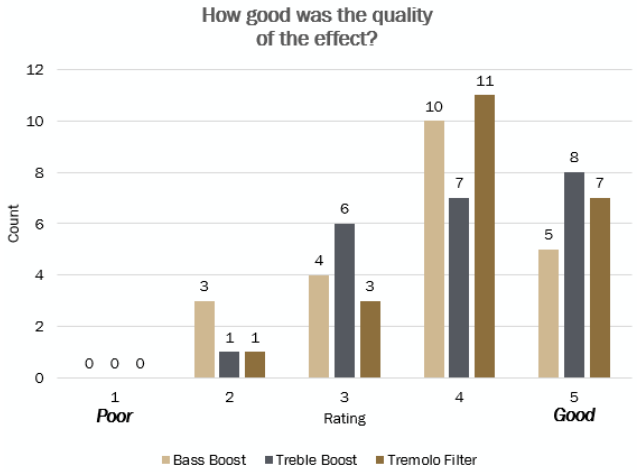

Figure 17: Survey Result for Question 2

IV. Member Contributions

Alex Chitsazzadeh

- **LED matrix drivers (matrix_driver.h/.c)**
- **SD Card wrapper functions (my_sdcard.h/.c)**
- **WAV parser functions (wav_parser.h/.c)**
- **FFT driver functions (fft_driver.h/.c)**
- **DAC output functions (dac_output.h/.c)**
- **Circular buffer functions (circular_buffer.h/.c)**
- **Audio Processing Tremolo Filter in time domain (audio_processing.h/.c)**
- **Main program structure and loop (main.h/.c)**
- **Collected audio output data from STM32 and plotted FFT graphs in Python to compare before/after filter effects**

Aditya Sood

- **LED matrix drivers (matrix_driver.h/.c)**
- **Prototyped and debugged SD card interfacing with FatFs through SPI and SDIO peripherals**
- **Collected survey data for evaluation**
- **Edited project demo video**

Kerway Tsai

- **Prototyped, debugged, and tested various LM386 amplifier circuits**
- **Designed and planned hardware schematic and pinout**
- **Collected research on existing solutions**

Vinay Jagan

- **OLED drivers (oled_driver.h/.c)**
- **OLED drivers (main.h/.c)**

Derek Xavier Rosales

- **LED matrix interpolation (matrix_driver.h/.c)**
- **FFT driver functions (fft_driver.h/.c)**
- **Audio Processing Bass Boost in time domain and frequency domain (audio_processing.h/.c)**

Zichen Zhu

- **User input functions (user_inputs.h/.c)**
- **ADC configuration (main.h/main.c)**
- **Prototyped, debugged, and tested various LM386 amplifier circuits**

Austin Carlton Kinkade:

- **Audio Processing Treble Boost in time domain (audio_processing.h/.c)**

V. Conclusion

A. Limitations

The biggest limitation within this project was streaming the .WAV data from the micro-SD card. Unfortunately, FatFs forcibly blocks the CPU when a read request is sent to the micro-SD card until the transaction is completed. Even with DMA2 Stream3 and DMA2 Stream6 enabled as shown in *Figure 1*. the FatFs code still forces the CPU to wait until the DMA engine completes the read. The reasoning for this appears to be because the micro-SD card requires a specific kind of acknowledgment and a lot of overhead in FatFs needs to be taken care of. Clearly, this is difficult to manage on bare-metal when no real operating system is present, so the FatFs driver opts to block the CPU until the read transaction is fully complete. SD card read times, like any secondary disk storage, come with a variable length latency. We experimentally determined that reading samples in larger bursts appears to be the most efficient way to exit the FatFs layer as quickly as possible. Reading 2048 bytes (1024 samples) seemed to be the sweet spot, so that is what we operated on. Unfortunately, this now means that we would need to perform a 1024-pt FFT if we were to process this data in the frequency domain, which proved to take too much time. Many sacrifices had to be made in order to keep up with the CPU-blocking reads, such as limiting how often the OLED frame is redrawn, limiting how many interpolation steps the LED Matrix can do, and limiting the scope of our "Audio Effects" to a few simple first-order IIR filters in the time-domain, rather than using the frequency domain approach. We implemented and tested a bass boost filter via the FFT/IFFT method, and it wound up being too slow for our audio streaming needs. In addition, doubling the playback frequency to 88.2 KHz would for sure be impossible to keep up with, so we were unable to implement effects that altered the playback speed of the song due to limited performance.

A solution to this limitation is to introduce a second hierarchy of memory that is smaller than the micro-SD card, but faster overall. The idea is to have an initial loading phase before the song starts where the selected .WAV file is transferred from the micro-SD card to the faster memory. Unfortunately, since

we are playing .WAV files, which are completely uncompressed, file sizes are too large to load onto the STM32F413ZH's local flash memory or RAM. About ~16MB is needed just for a .WAV file that only lasts a few minutes. Perhaps purchasing an external QSPI flash memory IC that is large enough to hold most .WAV files (32MB+) could have been a solution, but by the time we realized this problem, it was too late to make such major changes. We could certainly set up a DMA engine to work by transferring the data from the external QSPI flash into the MCU without blocking the CPU since FatFs wouldn't be in the way. In the end, we were still able to work with and optimize what we had to meet our goals of seamless audio playback with a few filtering effects.

One more thing to note is that because of the variable length reads from the micro-SD card, it was sometimes observed in the debugger that the CPU would be ready to push new samples to the DAC before the DMA was finished transmitting the previous sets of samples. Sometimes the CPU would get ahead by many iterations, likely due to some kind of buffer caching system in the FatFs layer where certain strings of data are accessed very quickly. We solved this by implementing double buffering in the DMA as well as the circular buffer with the producer/consumer paradigm between the CPU and DMA1 Stream5.

B. Potential Future Works

For the software level, future work can firstly extend the support for multiple audio formats by porting open source libraries (such as libmad) on top of the existing .WAV playback to realize real-time playback of lossy/lossless files such as MP3, AAC, and so on. At the same time, with the help of CMSIS-DSP or self-programmed FIR/IIR kernels, higher-order equalizers and filters could be introduced to provide users with more fine-grain acoustic tuning options. The player can also incorporate "fast seek" indexing - building a frame table while parsing the file header - allowing users to instantly jump to any point in time and maintain audio continuity. Combined with a Wi-Fi module or Bluetooth module, the device can be connected wirelessly and controlled remotely from a mobile device, further enhancing ease of use and convenience.

For the hardware level, the next phase of the project could be to integrate the circuits currently scattered on breadboards onto a custom PCB: unified wiring, zoned grounding, and RF and audio isolation to reduce noise and improve reliability. The PCB can be pre-packaged with either ESP32 or STM32 for low-power Wi-Fi/Bluetooth dual-mode connectivity, and integrate a Class-D amplifier, Li-ion battery charging and discharging management, and a power meter and USBC interface on the board. Of course, building such a PCB with all these desired features is out of scope for this current project, but they could be interesting ideas for building up and iterating on our product.

C. Conclusive Summary

This project aims to finish a cost-effective digital audio player that integrates hardware and software to provide hobbyists with a simplified real-time signal processing platform that is not as expensive as existing commercial high-grade prodcuts. The system uses the STM32F413ZH microcontroller as the core, utilizing its FPU and ample SRAM to complete digital audio filtering and double-buffered audio streaming processing. Peripherals include an LED matrix for real-time spectrum visualization, an OLED screen for menu and progress display, five potentiometers and four buttons for user input, and an attempt to use an LM386 amplifier to drive the speakers. At the software level, FatFs is used to manage the files in the micro-SD card. Various techniques are used to optimize usage of the CPU, such as utilizing DMA engines for memory transfers between devices or utilizing circular buffers so that producers and consumers are not constantly tied-up waiting for each other. System performance was assessed with a demo video and a structured questionnaire completed by participants. Low-frequency (bass) enhancement proved to be both the most noticeable and the most favorably rated, with the high-frequency (treble) boost and tremolo filter close behind, confirming the effectiveness of the system.

REFERENCES

[1] "Activo P1," ACTIVO, https://www.activostyle.com/p1 (accessed Mar. 10, 2025).

[2] "JM21," Guangzhou Feiao Electronic Technology Co, https://www.fiio.com/jm21 (accessed Mar. 10, 2025).

[3] "SR35," Astell&Kern, https://www.astellnkern.com/product/product_detail.jsp?productNo=143 (accessed Mar. 10, 2025).

[4] "M23," Guangzhou Feiao Electronic Technology Co, https://www.fiio.com/m23 (accessed Mar. 10, 2025).

[5] "NW-A306 Digital Media Player," Sony Electronics, https://electronics.sony.com/audio/walkman-digital-recorders/walkman-mp3-players/p/nwa306-b?srsltid=AfmBOoqhjCuimn21k2wdUKMYFU3fn4kbDpQWmC1itmfmOt3pbzGFVepw (accessed Mar. 10, 2025).

[6] "FatFs," ChaN, http://elm-chan.org/fsw/ff/doc/appnote.html (accessed May. 2, 2025).

APPENDIX

[1] Digital Audio Effects Player GitHub: https://github.com/alexc0888/Digital-Audio-Effects-Player/tree/master

[2] Digital Audio Effects Player Demo Video: https://purdue0-my.sharepoint.com/:v:/r/personal/achitsaz_purdue_edu/Documents/Digital%20Audio%20Effects%20Player%20Project/Demo%20Footage/Final_Demo.mp4?csf=1&web=1&e=mvXJDT&nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D