Keyboard DJ Set

Introduction

Team Members:

- Manas Gandhi (manaspg2)
- Jack Prokop (jprokop2)
- Milind Sagaram (milinds2)

Problem

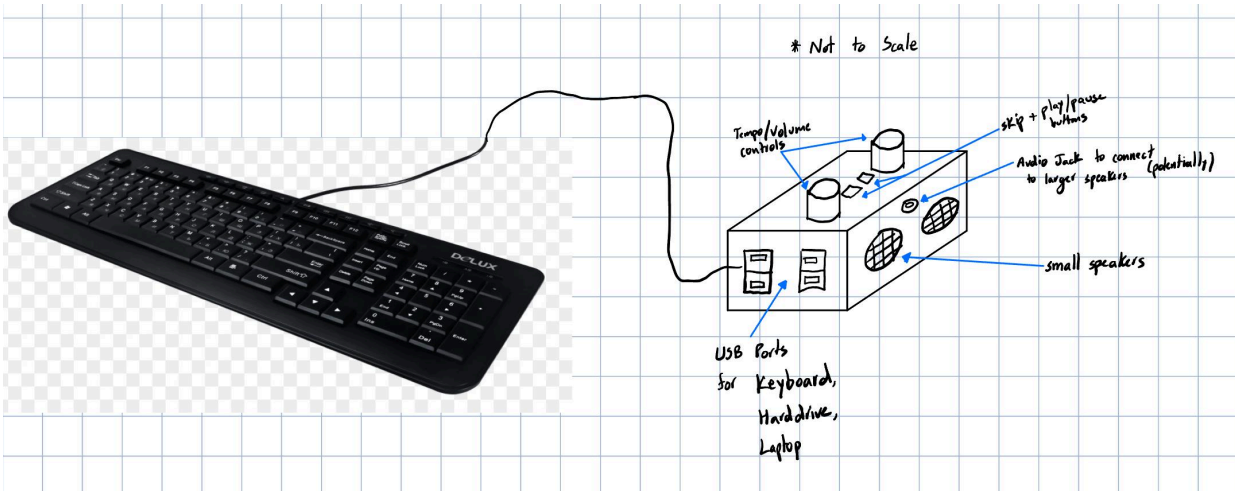
DJ boards have become the “hot topic” of today’s music industry, with the tool giving way to many of the greatest artists of our generation, including *John Summit* and *Twinsick*. While we have seen many great EDM artists shine due to the traditional DJ set, it has some drawbacks as well - namely the lack of portability, ease of use for new users, and high prices. First off, DJ boards tend to be large, heavy, and tough to transport. For DJs who want to go practice outside or DJ parties, this setup is suboptimal due the size and weight of the board. DJ boards are also incredibly complicated: there are two wheels, nearly a dozen knobs, a couple sliders, and many buttons on a board. For enthusiasts who want to learn the basics of DJ-ing, this creates a high learning barrier. Lastly, the cost of a DJ board is very high. Boards can be north of $300, which is out of budget for most people who are interested and want to learn the fundamentals of mixing music together. Hence, we propose a portable, easy to use, and cost effective DJ board to help young and interested DJs learn.

Solution

To address these challenges, we propose the DJ keyboard, a DJ board that simply uses the keys on an external keyboard, connected to a computer. This makes use of a microcontroller on a PCB for processing the inputs of the keyboard and converting that into commands for the software. We also will create software for the songs and audio processing, as well as the speaker technology. This approach simplifies the complicated DJ board, reduces the cost and size of getting a DJ board, and makes it easy to take anywhere, making the DJ experience for everyone much easier.

The specific DJ board elements we want to incorporate are volume control, tempo control, music slicing and looping, song skipping functionality, and if we have time, auto-crossfade capabilities

Visual Aid



The diagram outlines how the system is going to look as the finished product, with the keyboard connected to the encasing of the DJ board (which consists of the PCB and all the knobs. On a basic functionality level, the DJ board encasing

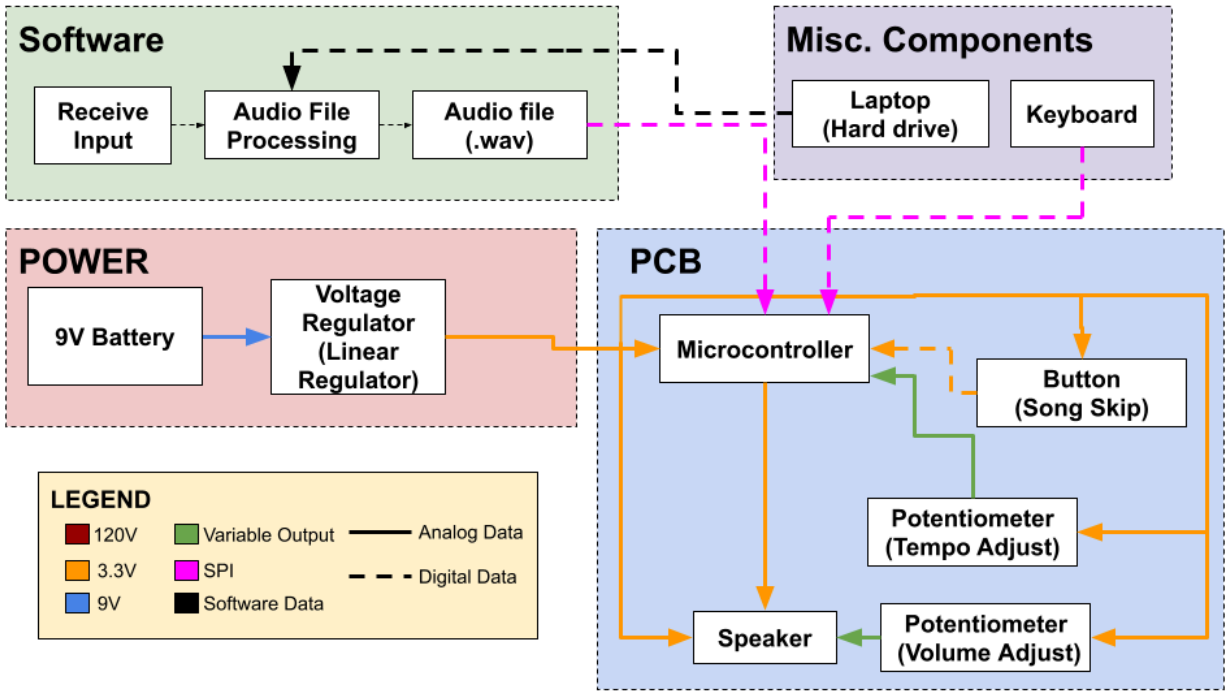
High Level Requirement List

1. **Simultaneous Track Playback:** Our DJ board must be capable of playing two tracks simultaneously, allowing users to transition between songs or mix them together. This feature is essential for replicating the core functionality of traditional DJ equipment, enabling smooth transitions and creative layering of audio tracks.
2. **Precision Tempo Control:** The DJ board must support the ability to increase or decrease the tempo of each track by at least 3 beats per minute (BPM). This level of precision will give users the flexibility to fine-tune the speed of the music, allowing for smooth beat-matching.
3. **Dynamic Volume Adjustment:** The DJ board must allow users to adjust the volume of each track by at least 10 decibels (dB) in both directions (increase or decrease). This

ensures that users can balance audio levels during live performances or practice sessions.

Design

Block Diagram



Subsystem Overview + Requirements

PCB

This subsystem will hold all the electrical components. The parts of this subsystem are shown below.

1. Microcontroller

This subsystem is the core of the whole system, providing the communication between the input (keyboard) and the software. This is where the processing of the keys will happen, as well as sending signals to the hardware.

The microcontroller will be responsible for receiving the keystroke/signals from hardware (in the form of button presses/potentiometer changes) and sending that information to the software. It will also be responsible for receiving audio files from the software and sending that data to the speaker to output the volume.

- **Communication between keyboard/hardware and microcontroller:** This is the first part of the communication that the microcontroller does. It receives inputs from the keyboard and hardware and translates that into information that is readable and usable by software. It will then send this information to the software to use.

- **Communication between software and microcontroller:** this is the second part of the communication that the microcontroller does. After the software does its processing, the software will send the microcontroller the audio files (.wav files) to send to the speaker to play.

3. Volume Control

This subsystem is where we control the volume using the PCB. This will be in the form of a potentiometer, which will send a signal to the microcontroller.

- **Potentiometer:** We will use a potentiometer (which we might connect to a slider) to increase and decrease the volume (increased resistance = lower volume). This will be directly connected in series with the speaker, controlling the volume output of the speaker.
- **Speaker:** We will put two small speakers on the PCB, and connect to output of the potentiometer to the input of the speaker to allow audio control.

4. Tempo Control

This subsystem is where we control the tempo of the song (in BPM) that is currently playing. This will also use a potentiometer, but it will send a signal to the microcontroller through an ADC that will let the microcontroller read voltage levels.

- **Tempo controller:** We will use a potentiometer (which we might connect to a slider) to send a signal to the microcontroller to increase or decrease the tempo. It will be connected to ground and voltage, of course, as well as an ADC that is connected to the microcontroller, so that the microcontroller can get the voltage output of the potentiometer and translate that to a tempo increase or decrease. We will have to set a default value as well (the potentiometer will be set to the middle to begin, and that will be +0 bpm).

5. Skip song

This subsystem is where we skip the song that is currently playing, allowing us to switch to the next song. This will use a button, connected to the microcontroller.

- **Skip button:** We will use a button that sends a high signal to the microcontroller, which will tell the software to send the next song into the speaker, effectively skipping this song and replacing it with the next song.
- **Inputs:** The keyboard should take inputs for all keys on the keyboard, and maybe combinations of keys (if we have time).

Power

The power subsystem is composed of two key components: a battery and a voltage regulator. Together, these elements ensure that each part of the system receives the appropriate power it needs for smooth operation. The battery serves as the main power source, and will be 9V. It will handle the charging and discharging to maintain reliable power output.

The voltage regulator plays a crucial role in distributing the correct voltage to different subsystems. For instance, the microcontroller requires a 3.3V power rail to function properly, so the power subsystem must supply exactly 3.3V to it. The rest of our components only require 3.3V, but if other components in the system require different voltage levels, such as 5V, the power subsystem must provide the correct voltage for those components as well. This ensures that each subsystem operates efficiently with the power it requires.

Software

This subsystem where the control and processing of the system happens. The inputs should be processed into some functionality based on a DJ board here.

- **Receiving Input:** Inputs will be processed and translated into functions here, from keyboard/hardware to the microcontroller to the software.
- **Audio processing:** audio processing will occur here, where the audio files will be processed and manipulated based on the inputs of the keyboard.
    - Will have to incorporate signal processing libraries.
- **Audio file (.wav):** The .wav file will be the processed audio file that will be sent to the microcontroller.
- **Communication with Keyboard:** embedded software will be written to communicate with the microcontroller through a bidirectional USB using SPI protocol. This will be how the software will receive input data and send audio files to the microcontroller.

Misc. Components

1. Keyboard

This subsystem has the inputs that our system will be receiving. This will be in the form of keystrokes, which the microcontroller will go on to translate into different functions.

2. Laptop

This subsystem will be used for all the components we cannot buy, specifically for a hard drive and a speaker.

- **Hard drive:** hard drive is just for storage purposes, real DJ boards have hard drives that they plug into the boards typically

Tolerance Analysis

Microcontroller

The ESP32-S3 operates at 3.3V with a maximum current of around 500 mA. To ensure proper functionality, we need to make sure the power supply is constantly delivering 3.3V ± 5%. As such, as the power supply's voltage tolerance is ±5% of 3.3V = 3.135V to 3.465V.

Resistors

These resistors will be used for pull-up/down functions and setting reference voltages.
A 10kΩ resistor with a tolerance of ±5% means the actual resistance could vary between 9.5kΩ and 10.5kΩ. This will be fine for the design, as the functionality of pull-up/down of the resistor will not be affected, as the high voltage and low voltage will still have a clear difference.

Capacitors

The capacitors will be used for filtering and for other mechanisms. With a nominal value of 1μF and ±10% tolerance, the capacitance will range between 0.90μF and 1.10μF.

Keyboard

A typical USB keyboard consumes about 100mA. The ESP32-S3 and the power system need to be able to handle this load without significant voltage drops or instability.

Potentiometers

With a 10kΩ potentiometer, the tolerance will be ±10%, meaning the range is from 9kΩ to 11kΩ. This range will not critically affect the system, as the volume and tempo will only change slightly, to an unnoticeable amount.

Power System (9V Battery and Voltage Regulator

The 9V battery supplies power and the voltage regulator steps it down to 3.3V for the microcontroller and other components. These components will require the most tolerance, as they tend to vary in output. A typical 9V battery will range from 9V to around 6V when depleted. The voltage regulator should be able to provide a steady 3.3V output across this input range. The AZ1117 voltage regulator has a dropout voltage of about 1.15V [2], so it requires at least 4.45V input to maintain 3.3V output.

Push Buttons

These buttons are going to be debounced in software or with capacitors, and the current ratings are 160mA. This has a tolerance of ±5%, meaning the current rating can be between 152mA and 168mA, which is supported by the microcontroller GPIO pins, so we do not need to worry.

Software Tolerance Analysis

Another perspective to examine, tolerance wise, is the software, and the tolerance we need to be able to accomplish our goals with the software. Our software will adjust the tempo of a .wav file sampled at 44.1 kHz on an ESP32-S3 microcontroller. This must be completed within 0.5 seconds from the time the tempo adjustment request is initiated. The software will be run on an ESP32-S3 microcontroller, which has a 240 MHz clock speed, 384 KB ROM, and 512 KB SRAM. To quantify the analysis, we need to first define the parameters associated with the microcontroller and other data that the software will interact with.

Key Parameters for ESP32-S3

I. **Clock Speed:** 240 MHz (0.24 GHz)

II. **Memory Constraints:** 512 KB SRAM and 384 KB ROM

III. **WAV File Parameters:**

III.I **Sampling Rate**: 44.1 kHz (44,100 samples per second).

III.II **Bit Depth:** 16 bits (2 bytes) per sample.

IV. **Tempo Adjustment Requirement**: The system must be able to change the tempo of the audio and process it in real-time with a response time of less than 0.5 seconds.

We need to first consider the processing and memory part of the software. In terms of the processing, we can measure this by the complexity of the algorithm. The complexity of an FFT-based phase vocoder is O(n*log(n)), where n is the number of samples processed per frame. The FFT phase vocoder is how we process audio data. In terms of memory, we only have 512

KB SRAM in the microprocessor. This limits the number of samples that can be stored and processed in RAM at one time, and also impacts how large each audio processing frame can be. We also have 384 KB ROM that we can use to store program code and static data.

Given that each sample in the .wav file is 2 bytes (16-bit audio), the available 512 KB SRAM can hold:

$$(512 \times 1024)/2 = 262144 \text{ samples}$$

At a sampling rate of 44.1 kHz, this gives us:

$$262144 / 44100 \approx 5.944 \text{ seconds of audio that we can store at a time}$$

This means that the ESP32-S3 can hold about 6 seconds of uncompressed audio in memory at once, allowing room for buffering and overlap processing.

Now let's focus on the latency and processing speed. The key parameters to consider for this are:

I. **Frame Size (F):** the number of samples processed in one frame

II. **Algorithm Time Complexity (C(n)):** The complexity of the time-stretching algorithm. For the phase vocoder, it's O(n*log(n)), as we calculated before.

III. **Processing Speed (S):** The ESP32-S3 runs at a speed of 240 MHz.

IV. **Memory Bandwidth (B):** The speed at which the ESP32-S3 can access and process memory.

V. **I/O Latency (L_{IO}):** The time delay caused by reading audio from external flash memory and writing it back to the audio output device.

Let's first break down the processing time needed per frame, which we can call $T_{process}$. The processing time will be based on the time complexity and clock speed, giving us:

$$T_{process} = C(n) \times S$$

where n is the number of samples per frame. This is thus the tolerance for the processing time needed per frame. In terms of memory and I/O latency, the latency is based on the speed at which data can be read from or written to memory. For tolerance, we want to minimize this latency. For example, with real-time response time, the total time to process one frame (processing time + I/O latency) must allow for real-time tempo changes within 0.5 seconds. So:

$$T_{total} = T_{process} + L_{IO} \text{ and}$$

$$n \times T_{total} \leq 0.5 \text{ seconds}$$

So if we put all this together, we can get the total time taken per frame, to check our tolerance. Our processing time will be, as mentioned before, will be based on algorithmic complexity and processing speed of our microcontroller.

$$T_{process} = 240 \times 1{,}061{,}024 * \log_2(1024) = 240 \times 1{,}061{,}024 \times 10 \approx 4.26 \times 10^{-5}\ _{seconds} = 42.6\ \mu s$$

Our I/O latency per frame can be calculated assuming we have an I/O bandwidth of 20 MB/s. We also know that each frame has 1024 samples, and we have 2 bytes of sampling depth, so we get a data size of 2048 (2 KB) per frame. Hence:

$$L_{IO} = 2048\ _{bytes}/(20 \times 106_{(bytes/second)}) \approx 1.024 \times 10^{-4}\ _{seconds} = 102.4\ \mu s$$

Putting this all together, we can get the total time per frame:

$$T_{total} = T_{process} + L_{IO} = (4.26 \times 10^{-5}) + (1.024 \times 10^{-4}) \approx 1.45 \times 10^{-4}\ _{seconds} = 145\ \mu s$$

So based on all this, we can figure out that the frames that we can process within the 0.5 seconds we have are:

$$FPS = 0.5\ _{seconds}/T_{total} = 0.5\ _{seconds} / 1.45 \times 10^{-4}\ _{seconds/frame} \approx 3448\ _{frames}$$

Each frame contains 1024 samples, so the system can process:

$$\text{Samples} = \text{FPS} \times \text{samples} = 3448 \times 1024 \approx 3{,}529{,}152 \text{ samples}$$

Given that our sample rate is 44.1 kHz, the total time of audio processed is:

T_{net} = Samples / sample rate = 3,529,152 / 44,100 ≈ 80 seconds worth of audio data processed in 0.5 seconds.

This means that within 0.5 seconds, our system can only process 80 seconds worth of audio data from the .wav file, so we need to make sure the software has enough time to process all the audio files accordingly to different changes.

Ethics and Safety

Privacy and Data Security (ACM Code 1.6):

- **Issue:** The software might need to interact with personal files on a user's device, including music libraries. Our design must prevent unauthorized access to the user's music or other personal files.
- **Solution:** We will not collect or store any personal data from users unless explicitly required (such as music) and will follow strict data security measures (ex. consent).

Intellectual Property (ACM Code 1.5):

- **Issue:** Users could potentially use the software to play/mix copyrighted music without authorization, violating intellectual property laws.
- **Solution:** We will include disclaimers encouraging users to comply with copyright regulations and offer placed to legally obtained music files that can be used.

Honesty and Integrity (IEEE Code 7.8.I):

- **Issue:** Ethical guidelines also mandate transparency in communication. We must avoid misleading claims about the functionality or features of the DJ set.

- **Solution:** All documentation, including advertising or promotional materials, will clearly represent the capabilities of the system.

Tolerance Analysis for Audio Processing on ESP32-S3: Tempo Adjustment of a 44.1 kHz WAV File

Key Parameters for ESP32-S3

1. **Clock Speed**: 240 MHz (0.24 GHz).
2. **Memory Constraints**: 512 KB SRAM and 384 KB ROM.
3. **WAV File Parameters**:
    - **Sampling Rate**: 44.1 kHz (44,100 samples per second).
    - **Bit Depth**: 16 bits (2 bytes) per sample.
4. **Tempo Adjustment Requirement**: The system must be able to change the tempo of the audio and process it in real time with a response time of no more than 0.5 seconds.

Processing and Memory Considerations

1. Algorithm Choice

- **Phase Vocoder** or **Waveform Similarity Overlap Add (WSOLA)** algorithms are often used for real-time time-stretching (tempo adjustment) without affecting pitch. These algorithms are computationally complex and may be challenging to implement efficiently on a constrained microcontroller.

- The complexity of an FFT-based phase vocoder is **O(n log n)**, where n is the number of samples processed per frame. WSOLA is more memory-intensive due to the overlap-add method.

2. Memory Constraints

- **512 KB SRAM**: Limits the number of samples that can be stored and processed in RAM at one time. This impacts how large each audio processing frame can be.
- **384 KB ROM**: Available for storing program code and static data.

Given that each sample in the .wav file is 2 bytes (16-bit audio), the available **512 KB SRAM** can hold:

512×10242≈262,144 samples\frac{512 \times 1024}{2} \approx 262,144 \text{ samples}2512×1024≈262,144 samples

At a sampling rate of 44.1 kHz, this equates to:

26214444100≈5.94 seconds of audio\frac{262144}{44100} \approx 5.94 \text{ seconds of audio}44100262144≈5.94 seconds of audio

This means that the ESP32-S3 can hold nearly 6 seconds of uncompressed audio in memory at once, allowing ample room for buffering and overlap processing.

Key Variables for Tolerance Analysis

To conduct the tolerance analysis, we identify the main factors influencing system performance on the ESP32-S3:

1. **Frame Size (F)**: The number of samples processed in one frame.
2. **Algorithm Time Complexity (C(F))**: The complexity of the time-stretching algorithm (e.g., Phase Vocoder's **O(F log F)**).
3. **Processing Speed (S)**: The ESP32-S3's 240 MHz clock speed.
4. **Memory Bandwidth (B)**: The speed at which the ESP32-S3 can access and process memory.
5. **I/O Latency (L_io)**: Latency introduced by reading audio from external flash memory and writing back to the audio output device.

Tolerance Analysis Framework

1. **Processing Time per Frame (T_process)** The processing time per frame depends on the algorithm's complexity and the clock speed of the ESP32-S3:

   Tprocess=C(F)×FST_{process} = C(F) \times \frac{F}{S}Tprocess=C(F)×SF

   Where:

   - C(F) is the time complexity (assumed **O(F log F)** for an FFT-based algorithm).
   - F is the number of samples in a frame.
   - S is the ESP32-S3's processing speed (240 MHz or 240 million cycles per second).
2. **Memory and I/O Latency (L_io)** The ESP32-S3's I/O performance depends on how quickly data can be read from or written to external memory (such as SPI flash). Latency must be minimized, especially if the file is not stored entirely in SRAM.
3. **Real-Time Response Time** The total time to process one frame of audio (processing time + I/O latency) must allow tempo changes to happen within 0.5 seconds:

Ttotal=Tprocess+LioT_{total} = T_{process} + L_{io}Ttotal=Tprocess+Lio

To meet the real-time requirement, the total time to process all frames within 0.5 seconds must satisfy:

n×Ttotal≤0.5 secondsn \times T_{total} \leq 0.5 \text{ seconds}n×Ttotal≤0.5 seconds

Where n is the number of frames processed within 0.5 seconds.

Example Calculation

Let's calculate the processing time and assess whether the ESP32-S3 can adjust the tempo in real time.

Step 1: Processing Time per Frame

For an FFT-based phase vocoder, the time complexity is **O(F log F)**. Let's assume a frame size (F) of **1024 samples**.

- **Algorithm Complexity (C(F))**: Approximate **FFT complexity** is **F log F**.
- **Processing Speed (S)**: ESP32-S3 runs at **240 MHz**.

We can estimate the time required for processing one frame as:

Tprocess=1024log2(1024)240×106=1024×10240×106=4.26×10−5 seconds (42.6 microseconds)T_{process} = \frac{1024 \log_2(1024)}{240 \times 10^6} = \frac{1024 \times 10}{240 \times 10^6} = 4.26 \times 10^{-5} \text{ seconds (42.6 microseconds)}Tprocess=240×1061024log2(1024)=240×1061024×10=4.26×10−5 seconds (42.6 microseconds)

Step 2: I/O Latency per Frame

Assume the file is read from external SPI flash and the I/O bandwidth is **20 MB/s** (a reasonable estimate for the ESP32-S3's SPI flash access).

- **Frame size**: 1024 samples per frame, and each sample is 2 bytes.
- **Data size per frame**: 1024 samples × 2 bytes = 2048 bytes (2 KB).

The I/O latency for reading one frame is:

Lio=2048 bytes20×106 bytes/second=1.024×10−4 seconds (102.4 microseconds)L_{io} = \frac{2048 \text{ bytes}}{20 \times 10^6 \text{ bytes/second}} = 1.024 \times 10^{-4} \text{ seconds (102.4 microseconds)}Lio=20×106 bytes/second2048 bytes=1.024×10−4 seconds (102.4 microseconds)

Step 3: Total Time per Frame (T_total)

Ttotal=Tprocess+Lio=4.26×10−5+1.024×10−4≈1.45×10−4 seconds (145 microseconds)T_{total} = T_{process} + L_{io} = 4.26 \times 10^{-5} + 1.024 \times 10^{-4} \approx 1.45 \times 10^{-4} \text{ seconds (145 microseconds)}Ttotal=Tprocess+Lio=4.26×10−5+1.024×10−4≈1.45×10−4 seconds (145 microseconds)

Step 4: Frames Processed within 0.5 Seconds

We can now calculate how many frames can be processed within the 0.5-second window:

0.5 seconds1.45×10−4 seconds/frame≈3448 frames\frac{0.5 \text{ seconds}}{1.45 \times 10^{-4} \text{ seconds/frame}} \approx 3448 \text{ frames}1.45×10−4 seconds/frame0.5 seconds≈3448 frames

Each frame is **1024 samples**, so in total, the system can process:

3448×1024≈3,529,152 samples3448 \times 1024 \approx 3,529,152 \text{ samples}3448×1024≈3,529,152 samples

Given that the audio is sampled at 44.1 kHz:

3,529,15244100≈80 seconds of audio processed in 0.5 seconds (with tempo adjustment)\frac{3,529,152}{44100} \approx 80 \text{ seconds of audio processed in 0.5 seconds (with tempo adjustment)}441003,529,152≈80 seconds of audio processed in 0.5 seconds (with tempo adjustment)

Conclusion

- **Key Finding**: The ESP32-S3 can process approximately **3.5 million samples** within the required **0.5-second window**, far exceeding the number of samples that would typically be required for real-time tempo adjustment of a 44.1 kHz WAV file.
- **Tolerance**: The system has a significant processing buffer, with the ability to process **80 seconds of audio** for each half-second of real time, providing considerable tolerance for variations in workload or hardware performance.
- **Memory Constraints**: The **512 KB SRAM** is sufficient to hold nearly **6 seconds of uncompressed audio** (262,144 samples), ensuring that sufficient audio can be stored for overlap-add methods or FFT-based processing.
- **I/O Bottlenecks**: The primary bottleneck in the system is I/O latency when accessing external SPI flash memory. Optimizing the I/O subsystem (e.g., by reducing the

frequency of memory access or improving buffering) will help further ensure that the system remains within the 0.5-second real-time requirement.

The ESP32-S3, despite being a resource-constrained microcontroller, is capable of handling real-time tempo adjustments for audio at 44.1 kHz, provided that the time-stretching algorithms are optimized for embedded processing.