Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет __Комп'ютерної інженерії та управління_____
(повна назва)

Кафедра Автоматизації проектування обчислювальної техніки
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти____перший (бакалаврський)_____
(рівень вищої освіти)

Система створення аудіоефектів на платформі Raspberry Pi
(тема)

Виконав:
здобувач ____4_____ року навчання,
групи КІУКІ-21-7

_____Степанян Є.С._____
(прізвище, ініціали)

Спеціальність *123 _ Комп'ютерна інженерія*

Тип програми ____освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма_____

_____*Комп'ютерна інженерія*_____
(повна назва освітньої прграми)

Керівник _ас. Корнієнко В.Р._
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ _Чумаченко С.В
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерної інженерії та управління _____

Кафедра_____Автоматизації проектування обчислювальної техніки_____

Рівень вищої освіти____перший (бакалаврський)_____

Спеціальність_____123_Комп'ютерна інженерія _____
(шифр і назва)

Тип програми _____Освітньо-професійна_____
(освітньо-професійна або освітньо-наукова)

Освітня програма _____Комп'ютерна інженерія _____
(повна назва

ЗАТВЕРДЖУЮ:

Зав. кафедри ____  ____
(підпис)

«__02__»___05_____ 2024 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві_____Степаняну Євгенію Сергійовичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи (проекту Система створення аудіоефектів на платформі Raspberry Pi_____
_____

затверджена наказом по університету від від " 21 "_____05_____2025 р. № 403 Ст_____

2. Термін подання здобувачем роботи до екзаменаційної комісії _____10.06.2025_____

3. Вихідні дані до роботи (проекту)_____

Апаратна платформа Raspberry Pi

Мова програмування C_____

Кросс-компіляція_____

Embedded Linux_____

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

Цифрова обробка сигналів

Математичні моделі аудіо ефектів

Ааудіо-підсистеми апаратної платформи Raspberry Pi

Бібліотеки для реалізації потокової обробки аудіосигналів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) ___13 слайдів_
_____
_____
_____
_____
_____

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи (проекту)	Термін виконання етапів проекту (роботи)	Примітка
1	Видача теми проекту, узгодження і затвердження теми	02.05.2025 -05.05.2025	
2	Аналіз проблемної галузі, постановка задачі, вибір інструментальних засобів	05.05.2025 -10.05.2025	
3	Аналіз технологій проектування на апаратній платформа Raspberry.	10.05.2025 -17.05.2025	
4	Аналіз аудіо-підсистеми апаратної платформи Raspberry Pi	18.05.2025 -25.05.2025	
5	Реалізація проекту генератора аудіоефектів	25.05.2025 -30.05.2025	
6	Фізична реалізація проекту	31.05.2025 -05.06.205	
7	Оформлення пояснювальної записки	05.06.2025 -10.06.2025	
8	Перевірка виконаного проекту керівником,	10.06.2025 -15.06.2025	
9	Захист проекту	15.06.2025 -20.06.2025	

Дата видачі завдання 02.05.2025_____

Здобувач _____
(підпис)

Керівник роботи _____ ас Корнієнко В.Р.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Записка пояснювальна: 48 сторінок, 15 рисунків, 7 джерел за переліком посилань.

ЦИФРОВА ОБРОБКА СИГНАЛІВ, ВБУДОВАНІ СИСТЕМИ, RASPBERRY PI, КРОСС-КОМПІЛЯЦІЯ, МОВА ПРОГРАМУВАННЯ C++

Метою кваліфікаційної роботи є розробка системи створення аудіоефектів на платформі Raspberry Pi.

Система реалізована з використанням апаратної платформи Raspberry Pi 3B+, в якості засобу відображення параметрів ефектів та зміни їх властивостей використовується TFT SPI дисплей приєднаний до плати. Для виконання виводу звукового аудіо-потоку використовується вбудована аудіо-підсистема платформи Raspberry Pi. Реалізація проекту виконана на базі інструментального стеку VSCode, CMake, Docker для кросс-компіляції проекту і з використанням мови програмування C++20.

ABSTRACT

Explanatory note: 48 pages, 15 figures, 7 sources according to the list of links.

DIGITAL SIGNAL PROCESSING, EMBEDDED SYSTEMS, RASPBERRY PI, CROSS-COMPLICATION, C++ PROGRAMMING LANGUAGE

The goal of the qualification work is to develop a system for creating audio effects on the Raspberry Pi platform.

The system is implemented using the Raspberry Pi 3B+ hardware platform. A TFT SPI display connected to the board is used to display the effect parameters and change their properties. The built-in audio subsystem of the Raspberry Pi platform is used for audio stream output. The project is implemented using the tool stack VSCode, CMake, Docker for cross-compilation, and the C++20 programming language.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

АЦП – аналого-цифровий перетворювач;

LFO – low frequency oscillator;

КІХ – кінцева імпульсна характеристика;

ЦАП – цифро-аналоговий перетворювач;

ШПФ – швидке перетворення Фур’є;

API – application program interface;

DSP – digital signal processing;

IDE – integrated development environment;

SBC – single board computer;

IR – Impulse response

SD -serial data

WS-word select

CLK-clock

ВСТУП

Проблема обробки аудіосигналів залишається актуальною в умовах стрімкого розвитку цифрових технологій. Зростаючі вимоги до якості звуку в мультимедійних додатках, мобільних пристроях, системах віртуальної та доповненої реальності, а також в індустрії розваг зумовлюють необхідність створення гнучких, ефективних та ресурсозберігаючих систем аудіообробки.

Системи аудіо ефектів відіграють важливу роль у формуванні акустичного простору, корекції звукових характеристик та забезпеченні креативної складової звукового контенту. Використання таких ефектів дозволяє покращити сприйняття сигналу, створити ілюзію простору, додати глибину або виділити певні елементи аудіоматеріалу. До найпоширеніших аудіо ефектів належать реверберація, затримка (delay), еквалайзер, компресія, хорус, фленжер та інші. Їх застосування охоплює як професійне студійне середовище, так і побутові рішення для онлайн-комунікації чи потокової трансляції.

Особливий інтерес становить реалізація систем аудіо ефектів у рамках вбудованих або реального часу програмних рішень. Це зумовлено необхідністю створення компактного, швидкого та енергоефективного програмного забезпечення, здатного працювати на обмежених апаратних ресурсах. Саме тому актуальними стають дослідження та розробки, спрямовані на оптимізацію алгоритмів цифрової обробки сигналів, зниження затримок при обробці, а також забезпечення масштабованості та модульності реалізації.

У контексті сучасних тенденцій особливої уваги набувають програмно-апаратні системи на базі відкритих платформ, таких як Raspberry Pi, Arduino, STM32 тощо, а також розробка кросплатформених програмних рішень із використанням популярних мов програмування та бібліотек. Це дає змогу інтегрувати систему аудіо ефектів у ширший спектр застосунків — від

музичного програмного забезпечення до систем розпізнавання мовлення чи шумозаглушення.

Таким чином, реалізація системи аудіо ефектів є доцільним та актуальним напрямом, що поєднує теоретичні аспекти цифрової обробки сигналів із практичними задачами прикладного програмування. Створення такої системи дозволяє вивчити особливості побудови модульної архітектури, дослідити алгоритми обробки звуку, а також розробити функціональний продукт, що може бути застосований у реальних умовах.

1 МЕТОДИ СТВОРЕННЯ АУДІОЕФЕКТІВ ДЛЯ ЗВУКОВИХ СИГНАЛІВ

1.1 Репрезентація цифрового сигналу, теорема Найквіста

Коректний вибір частоти дискретизації є важливим аспектом цифрової обробки сигналів. Теорема Найквіста, також відома як теорема вибірки Найквіста-Шеннона, визначає мінімальну частоту дискретизації, необхідну для точного відновлення сигналу. Вона стверджує, що для коректного дискретного представлення аналогового сигналу його необхідно дискретизувати з частотою, яка як мінімум вдвічі перевищує максимальну частоту сигналу. Ця частота відома як частота Найквіста і визначається як:

$$f_s = 2f_{max}$$

Де f_s -часота дискретизації, а f_{max} максимальна частота сигналу.

Недотримання цієї умови призводить до явища, відомого як аліасинг, коли високочастотні компоненти сигналу спотворюються і накладаються на низькочастотні компоненти, що унеможливлює точне відновлення вихідного сигналу.

На рис. 1.1 наведено приклади дискретизації аналогового сигналу та аліасінгу

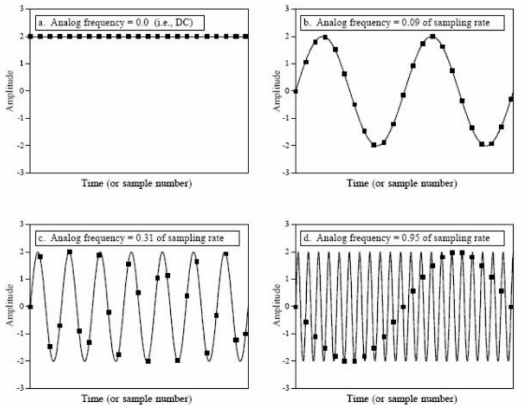

Рисунок 1.1 – Репрезентація процесу дискретизації аналогового сигналу та явища аліасінгу

.

Існує два терміни що широко використовуються при обговоренні теореми вибірки, а саме частота Найквіста та швидкість Найквіста. Їхнє значення не є стандартизованим. Для розуміння цього розглянемо аналоговий сигнал, що містить частоти між постійним струмом (DC) та 3 кГц. Щоб правильно оцифрувати цей сигнал, його необхідно дискретизувати з частотою 6 000 вибірок/с (6 кГц) або вище. Припустимо, що ми вибираємо частоту дискретизації 8 000 вибірок/с (8 кГц), що дозволяє правильно представляти частоти між DC та 4 кГц. У цій ситуації існують чотири важливі частоти:

- найвища частота в сигналі — 3 кгц;
- подвоєна ця частота — 6 кгц;
- частота дискретизації — 8 кгц;
- половина частоти дискретизації — 4 кгц.

Таким чином, в літературних джерелах є можливим зустріти обидва терміни, як Nyquist Frequency, так і Nyquist rate [1].

1.2 Теорія цифрових КІХ та БІХ фільтрів

У цифровій обробці сигналів використовуються два основні типи фільтрів: кінцевої імпульсної характеристики (КІХ) та безкінечної імпульсної характеристики (БІХ). Основна відмінність між ними полягає у способі реакції на вхідний імпульсний сигнал.

КІХ-фільтри мають обмежену довжину імпульсної характеристики, що означає, що після закінчення обробки сигналу вихідний сигнал припиняється через визначену кількість відліків. Такі фільтри є стійкими та не мають зворотного зв’язку, що робить їх зручними для реалізації. Їхнім недоліком є те, що для досягнення високої селективності потрібно використовувати велику кількість коефіцієнтів.

БІХ-фільтри, на відміну від КІХ-фільтрів, мають нескінченну імпульсну характеристику, оскільки вони використовують зворотний зв’язок.

Це дозволяє їм досягати високої селективності з меншою кількістю коефіцієнтів, однак вони можуть бути нестабільними. Через зворотний зв’язок БІХ-фільтри можуть накопичувати похибки, що може призводити до нестабільності у певних випадках.

1.3 Теорія CIC фільтрів

CIC (Cascaded Integrator-Comb) фільтри є класом цифрових фільтрів, що використовуються для зміни частоти дискретизації сигналу без використання множень і у задачах симуляції реверберації. Вони особливо ефективні для реалізації децимації (зменшення частоти дискретизації) та інтерполяції (збільшення частоти дискретизації) в цифровій обробці сигналів.

CIC-фільтри знайшли широке застосування в комунікаційних системах, радіотехніці та цифровій обробці аудіосигналів. Основна перевага CIC-фільтрів полягає в їх простій реалізації, яка не потребує операцій множення, що знижує апаратні витрати. Основними частинами CIC-фільтру складається є інтегруючий каскад (Integrator stage) який накопичує суму поточних і попередніх значень сигналу та гребінчастий каскад (Comb stage) який обчислює різницю між поточним значенням і значенням, що було отримано на певну кількість відліків раніше.

Рівняння для вихідного сигналу CIC-фільтра складається з інтегруючої та гребінчастої частини де інтегруюча частина має наступне рівняння

$$y[n] = y[n-1] + x[n]$$

і комбінаторна частина має наступне рівняння

$$z[n] = y[n] - y[n-R]$$

де R – коефіцієнт зміни частоти дискретизації.

Зазвичай CIC-фільтри реалізуються у вигляді каскаду з кількох інтегруючих та комбінаторних блоків, що дозволяє отримати фільтр із високим порядком та ефективним придушенням небажаних компонент

спектру. Основні властивості CIC-фільтрів це відсутність операцій множення, лише операції додавання і віднімання, проста реалізація на FPGA, DSP та ASIC та наявність нульових частотних компонент у кратних частотах децимації/інтерполяції, що потребує додаткового згладжування.

Основним недоліком CIC-фільтрів є нерівномірність АЧХ, яка призводить до ослаблення високочастотних компонент сигналу. Для компенсації цього ефекту часто застосовують коригувальні фільтри (compensation filters).

1.4 Теорія швидкого перетворення Фур’є

Перетворення Фур'є є одним із ключових інструментів у теорії обробки сигналів, оскільки воно дозволяє реалізувати перехід від часової області репрезентації сигналу до частотної . Це перетворення визначає, як сигнал можна подати у вигляді сукупності синусоїдальних та косинусоїдальних компонентів різних частот, амплітуд та фаз.

Безперервне перетворення Фур'є використовується для представлення неперервних сигналів, а його дискретна версія, дискретне перетворення Фур'є (ДПФ), застосовується для аналізу цифрових сигналів. Особливо важливим є швидке перетворення Фур'є (ШПФ), яке значно знижує обчислювальні витрати при обробці великих обсягів даних.

Перетворення Фур'є широко застосовується у фільтрації сигналів, аналізі спектра, стисненні даних і в багатьох інших областях цифрової обробки. Воно дозволяє ефективно виявляти частотні компоненти сигналу та оцінювати їхню інтенсивність.

Одним з обмежень перетворення Фур'є є його припущення про періодичність сигналу, що може призводити до ефекту витікання спектра, коли енергія одного частотного компонента розподіляється на сусідні частоти. Для зменшення цього ефекту використовуються віконні функції, такі як Хеннінга, Гаусса, Ханна та інші, які мають бути застосовані для

вхідного буферу сигналу перед виконанням перетворення Фур’є.

1.5 Математичні моделі аудіо-ефектів

1.5.1 Ефект Flanger

Ефект фленджера отримав свою назву від оригінального аналогового методу, коли два магнітофони працюють з невеликим розривом синхронізації, коли на фланці (кільце, яке тримає стрічку) котушки одного магнітофона накладається палець, і таким чином застосовується тиск, щоб злегка уповільнити стрічку. Інші варіації включають кругове тертя по фланцю. Ефект створює затримку, що змінюється в часі, яка виходить із синхронізації, а потім знову входить у синхронізацію з першим магнітофоном. Цей ефект порівнюють із шумом реактивного двигуна або звуком автоматичного ефекту wah-wah. Як би ви не визначали це, фленджер має унікальний звук, і цей ефект можна почути на численних записах [2].

В ефекті фленджера час затримки варіюється між 0 і близько 2-7 мС, хоча це дуже суб'єктивно; вищі значення створюють глибший ефект, але додавання високих рівнів зворотного зв'язку може спричинити чути пінгування або шум, схожий на зіпер. Це означає, що мінімальна затримка становить 0,0 мС, а максимальна глибина — близько 2-7 мС. На рис. 1.2 зображено топологію блоку модульованого ефекту на базі статичної лінії затримки.

Фленджер використовує уніполярну модуляцію від мінімального часу затримки нуль до максимального значення, а потім знову до нуля; ми називаємо це модуляцією min-up. Коли індекс читання все далі відходить від індексу запису на кожному інтервалі зразка, висота тону знижується. Коли індекс змінює напрямок і наближається до індексу запису, висота тону збільшується — це ефект Доплера на практиці.

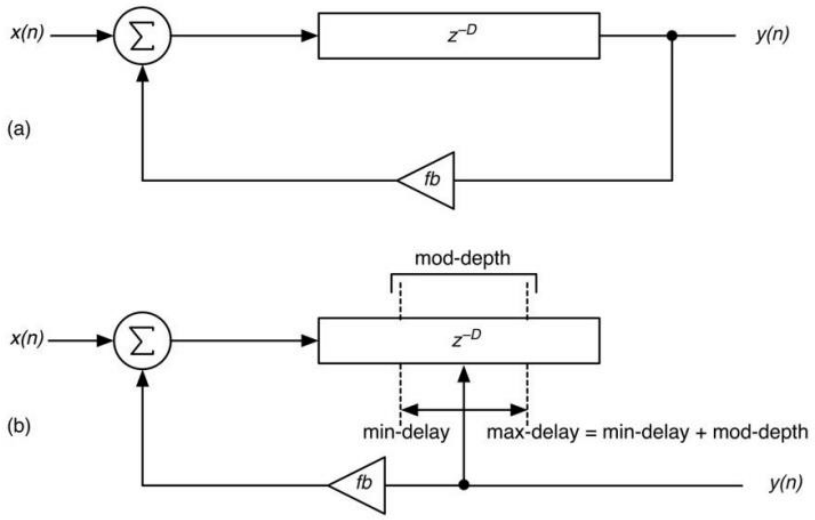

Рисунок 1.2 – Статична¶лінія затримки та модульований ефект затримки

Коли вихід змодельованої затримки виводиться на 100% wet- сигналом, без dry-сигналу та без зворотного зв'язку, ефект є вібрато: висота тону змінюється вверх і вниз, коли індекс читання рухається назад і вперед по діапазону затримки. Швидкість руху індексу читання контролює кількість зміщення висоти тону; швидші швидкості дають більше зміщення висоти тону, що найбільш помітно в ефекті вібрато. Форма хвилі LFO зазвичай є трикутною для фленджера та синусоїдальною для вібрато. Максимальна глибина для вібрато трохи вища, ніж для фленджера, але знову ж є суб'єктивним параметром. Зворотний зв'язок зазвичай не використовується з вібрато.

GUI елементи управління фленджером/вібрато зазвичай включають наступні налаштування:

– глибина: визначає, скільки з доступного діапазону часу затримки використовується;

– швидкість: швидкість LFO, що керує модуляцією;

– тип LFO: зазвичай синусоїдальна або трикутна, але може бути будь-яка;

– зворотний зв'язок або резонанс: кількість зворотного зв'язку в

системі, може бути позитивним або негативним (інвертований зворотний зв'язок) значенням:

– управління фленджером/вібрато: впливає на співвідношення dry/wet сигналу −100% dry= вібрато, 50/50 = фленджер.

1.5.2 Ефект Distortion

Оригінальні пристрої твердої фази спотворення, які використовують транзистори або діоди для виконання операції кліпування, не були спроектовані для імітації електронних ламп, а скоріше як унікальний ефект сам по собі. Згодом з'явилися інші пристрої, які намагалися звучати як електронні лампи. Ефект фузза — це більш радикальний електронний звук спотворення, який генерує високогейнгові квадратні хвилі, тоді як пристрої з нижчим рівнем підсилення, як-от Ibanez Tube Screamer® і Boss Super Overdrive®, використовують діоди для м'якшого кліпування. Tube Screamer використовує два діоди для створення симетричного кліпування, тоді як Super Overdrive використовує три для створення асиметричного кліпування. Більшість пристроїв складається з одного етапу спотворення, з помітним винятком оригінального Electro-Harmonix Big Muff Pi® distortion box, який використовував каскадуючу серію етапів спотворення, подібну до топології електронного лампового преамплера. Фільтр синтезатора Korg MS-20® включає вбудований діодний кліпер, який за своєю конструкцією дуже схожий на топологію Tube Screamer. Як і у випадку з електронними лампами, кліпер і фузз-платами є нелінійними пристроями, які додають гармоніки до вхідного сигналу. На рис 1.3 зображено деякі поширені схеми реалізації ефекту Distortion. Вибір топології, тип та кількість діодів/транзисторів і підсилення впливають на гармонічний контур отриманих спотворених виходів.

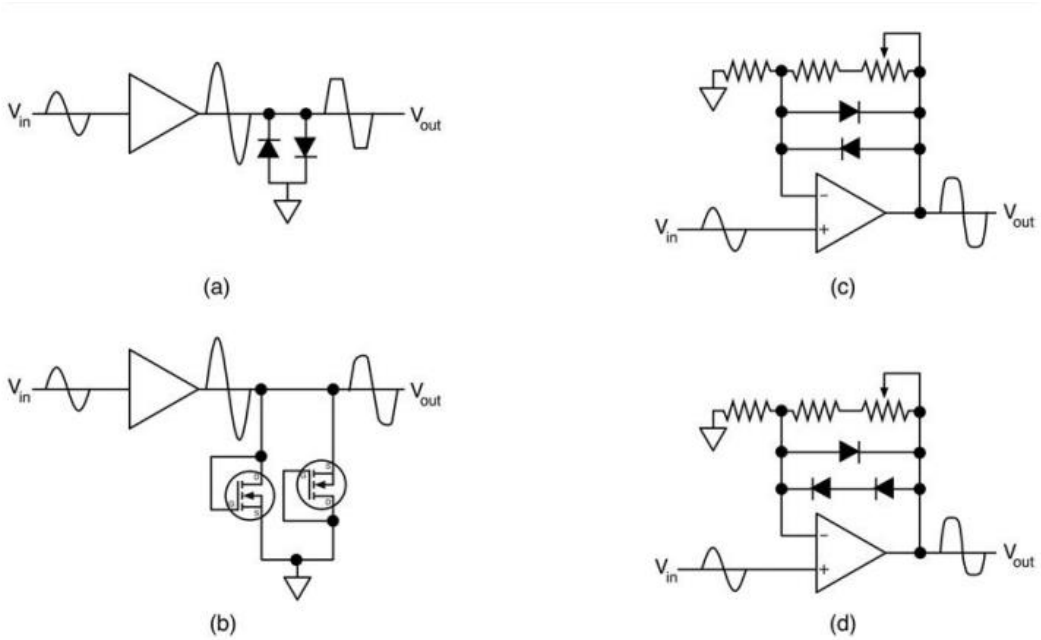

Рисунок 1.3 – Типові схемотехнічні топології реалізації ефекту “Distortion” на дискретних компонентах

1.5.3 Ефект реверберації

Ефекти реверберації мають універсальну привабливість, ймовірно, через те, що ми живемо в реверберaційному середовищі. Наші слухові органи є пристроями, що інтегрують час, використовуючи часові сигнали та транзієнти для отримання інформації, тому ми є чутливими до будь-яких змін, що маніпулюють цими сигналами. Існують два основних підходи до створення ефекту реверберації:

- реверберація через пряму згортку — фізичний підхід,
- реверберація через симуляцію — перцептуальний підхід.

У фізичному підході імпульсна відповідь приміщення згортатиметься з вхідним сигналом через довгий FIR фільтр. Для великих приміщень імпульсні відповіді можуть тривати від 5 до 10 секунд. У середині 1990-х років Гарднер розробив гібридну систему для швидкої згортки, що поєднувала пряму згортку з обробкою блоків FFT. Приблизно в той самий час Рейлі та МакГрат описали нову комерційну систему, що могла обробляти FIR фільтри з 262 144 тапами для згортки імпульсів тривалістю понад 5 секунд. Навіть за умови використання найефективніших методів швидкої

згортки, необхідна обчислювальна потужність часто буває занадто великою для практичного використання в плагінах. Окрім обчислювальних витрат, ще одним недоліком цього підходу є те, що імпульсна відповідь є зафіксованою в часі і відображає лише одну точку в приміщенні за певних умов. Ревербератор (або пристрій реверберації), здатний забезпечити багато різних реверберацій для різних просторів, вимагатиме великої бібліотеки файлів IR. Крім того, його параметри не можна легко коригувати.

Перцептуальний підхід намагається симулювати реверберацію з достатньою якістю і надати сприйняття реальної реверберації, при цьому забезпечуючи значно менші витрати на обробку сигналу. Переваги цього підходу очевидні — від мінімальних обчислювальних витрат до можливості змінювати численні параметри в реальному часі. Існує низка ключових інженерів, які зробили значний внесок у теорію, що використовується й сьогодні. Серед них — початкові роботи Шредера в 1960-х роках, а також подальші дослідження і внески Мура, Грізінгера, Гержона, Гарднера, Джота, Шейна, Сміта, Росчесо та інших протягом десятиліть. Багато матеріалів цього розділу є результатом їхнього внеску в цю галузь. Грізінгер зазначає, що неможливо досягти ідеальної імітації природної реверберації реального приміщення, і тому алгоритми завжди будуть наближеннями. Враховуючи це, галузь проєктування реверберацій має один із найбільш емпіричних підходів, заснованих на методах проб і помилок, серед всіх напрямків обробки аудіосигналів. На рис.1.4 зображено варіант топології ефекту реверебераці з використанням існуючої імпульсної характеристики кімнати.

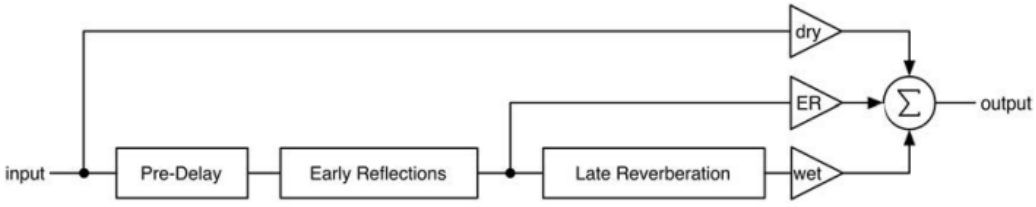

Рисунок 1.4 – Варіант реалізації ефекту реверберації з використанням існуючої імпульсної характеристики кімнати

1.6 Постановка задачі

Завданням даної роботи є розробка офлайн-системи обробки аудіо, що функціонує на апаратній платформі Raspberry Pi. Система повинна реалізовувати фіксований набір із чотирьох аудіо ефектів: біквратний фільтр низьких частот, ревербератор, дісторшен та фланжер, а також забезпечувати регулювання майстер-гучності. Обробка здійснюється у пакетному режимі — на вхід подається аудіофайл, після чого обробляється ланцюгом ефектів і відтворюється у гучномовцях.

Реалізація має враховувати обмежені ресурси Raspberry Pi, що потребує ефективного використання пам’яті та обчислювальних потужностей. Для забезпечення переносимості та можливості розробки на продуктивнішій системі передбачається використання засобів крос-компіляції.

2 МЕТОДИ СТВОРЕННЯ АУДІОЕФЕКТІВ ДЛЯ ЗВУКОВИХ СИГНАЛІВ

2.1 Розгляд аудіо-підсистеми апаратної платформи Raspberry Pi

Raspberry Pi — це компактний SBC(single board computer) який широко використовується в освіті, автоматизації, робототехніці та вбудованих системах. Основні характеристики останніх моделей включають до себе процесор з ARM-архітектурою з 4-8 ядрами, LPDDR4 оперативну пам’ять, графічний процесор з підтримкою відеовиходу через HDMI, аналоговий та цифровий аудіовихід.

Аудіо підсистема Raspberry Pi складається з наступних компонентів:

– ЦАП (Цифро-аналоговий перетворювач) – використовується для аналогового аудіовиходу через 3,5 мм роз’єм (у старих версіях);

– PWM-аудіо – широтно-імпульсна модуляція може бути використана для генерації звуку без використання окремого ЦАП.

– I2S-інтерфейс який дозволяє підключати зовнішні аудіомодулі (DAC, підсилювачі)

– HDMI-аудіо – передача цифрового звуку через HDMI-роз’єм для підключення до зовнішніх пристроїв відтворення.

– USB-аудіо з підтримкою зовнішніх USB звукових карт для покращення відтворення та запису аудіо.

Зовнішній вигляд міні-комп’ютера Raspberry Pi 3B+ представлений на рис.2.1:

Рисунок 2.1 – Міні-комп’ютер Raspberry Pi 3B+

2.2 Бібліотеки для реалізації потокової обробки аудіосигналів

«JUCE» — це бібліотека класів C++ для розробки кросс-платформених додатків, інтерфейсів користувача та компонентів де необхідність реалізувати обробку звуку. Він призначений для розробників, які створюють великі, складні програми на C++ і бажають використовувати лише єдиний високорівневий API замість різних бібліотек для різних цілей або платформ. Його функції включають компоненти з аудіо ефектами, класи для реалізації рядків, стандартні контейнери для даних, реалізацію роботи з XML, файловим вводом/вивідом, обмін повідомленнями та черги подій введення/виведення аудіо з низькою затримкою за допомогою CoreAudio, DSound і ASIO, аудіо буфери та вузли, маніпулювання файлами MIDI та подіями[3].

2.3 Огляд Visual DSP++

VisualDSP++® — це просте в установці та використанні інтегроване

середовище розробки та налагодження програмного забезпечення (IDDE), яке забезпечує ефективне керування проектами від початку до кінця з єдиного інтерфейсу.

Оскільки розробка проекту та налагодження інтегровані, є можливість швидко й легко переходити між редагуванням, створенням і налагодженням аудіо топології сигналу. Основні функції включають власний компілятор C/C++, розширені графічні інструменти побудови графіків, статистичне профілювання та ядро

VisualDSP++ (VDK), яке дозволяє реалізовувати код користувача більш структуровано та легше для масштабування. Інші функції включають асемблер, компонувальник, бібліотеки середовища розробки та cycle-accurate симулятори. VisualDSP++ є інструментом проектування аудіо топології, що значно скорочує час виходу продукту на ринок.

2.4 Огляд Sound Open Firmware

Sound Open Firmware (SOF) — це інфраструктура вбудованого програмного забезпечення та SDK з відкритим вихідним кодом цифрової обробки аудіосигналу (DSP). SOF надає інфраструктуру, елементи керування в реальному часі та аудіодрайвери як проект спільноти. Проектом керує Технічний керівний комітет звукової відкритої прошивки (TSC), до якого входять видатні та активні розробники з спільноти. SOF розробляється публічно та розміщено на платформі github.

Прошивка та SDK призначені для розробників, які зацікавлені в обробці аудіо або сигналу на сучасних DSP. SOF надає структуру, за допомогою якої розробники аудіо можуть створювати, тестувати та налаштовувати наступне:

- конвеєри обробки звуку та топології;
- компоненти обробки звуку;
- інфраструктура dsp та драйвери;

– інфраструктура хост-ос і драйвери.

На рисунку рис.2.2 зображено приклад топології реалізації еквалайзеру з керуванням з боку ОС параметрами еквалайзеру та загальною вихідною гучністю системи.

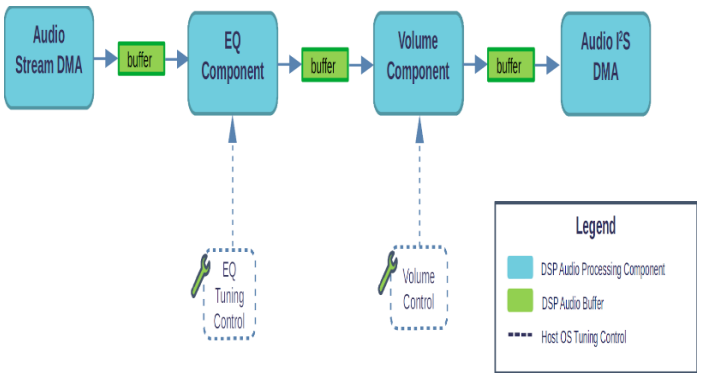

Рисунок 2.2 – Приклад топології еквалайзера з управлінням з боку ОС коефіцієнтами еквалайзера та вихідною гучністю системи.¶

3 РЕАЛІЗАЦІЯ СИСТЕМИ СТВОРЕННЯ АУДІО ЕФЕКТІВ НА ПЛАТФОРМІ RASPBERRY PI

3.1 Аналіз використаних технологій

Розробка вбудованих систем включає до себе вибір оптимальних і необхідних бібліотек що можуть зменшити час на розробку проекту і зменшити витрати на реалізацію рішень, що не є ключовими для основної тематики проекту. Зазвичай такі бібліотеки можуть включати до себе реалізації спеціалізованих структур даних, бібліотеки графічного інтерфейсу користувача, реалізацію обміну з периферійними блоками.

3.1.1 Бібліотека PFFFT

При постановці задачі реалізації ШПФ у проекті, зазвичай очевидним рішенням для проекту з відкритим вихідним кодом є використання бібліотеки FFTW. Однак це досить велика бібліотека, яка виконує все, що пов’язано з fft (2d перетворення, 3d перетворення, інші перетворення, такі як дискретно косинусне або швидке перетворення Хартлі). Ліцензією GNU GPL , що означає, що його не можна використовувати в продуктах без відкритого коду.

Альтернативою FFTW, яка має невеликий memory footpint, є FFTPACK v4, який доступний на NETLIB. Існує новіша версія (v5), але вона більша, оскільки має справу з багатовимірними перетвореннями. Це бібліотека, написана мовою FORTRAN 77, яку на поточний момент неможливо підтримувати з поточними інструментами. FFTPACKv4 був написаний у 1985 році доктором Полом Шварцтраубером з NCAR, більше 25 років тому. Незважаючи на свій вік, тести показують, що це все ще дуже ефективна бібліотека ШПФ. Згідно до тестів, які наявні у [4]

(http://www.fftw.org/speed/opteron-2.2GHz-32bit/) результати продуктивності реалізації перетворення показують високі результати.

Однак ця реалізація не конкурує з такими як FFTW, Intel MKL, AMD ACML, Apple vDSP. Причина цього полягає в тому, що ці бібліотеки використовують інструкції SSE SIMD, доступні на процесорах Intel, доступні з часів Pentium III. Ці інструкції стосуються невеликих векторів із 4 числами з плаваючою точкою одночасно, замість одного числа з плаваючою точкою для традиційного FPU, тому, використовуючи ці інструкції, можна очікувати 4-кратного підвищення продуктивності.

Реалізація PFFT полягає в тому, щоб перенести реалізацію з fortran fftpack v4, на C, змінити його для роботи з інструкціями SSE та перевірити, чи остаточна продуктивність може конкурувати з іншими бібліотеками SIMD FFT. Переклад на C було виконано за допомогою f2c (http://www.netlib.org/f2c/). Ця бібліотека має схожу до BSD ліцензію, яка сумісна з пропрієтарними проектами

3.1.2 Бібліотека mini audio

Бібліотека miniaudio є відкритою бібліотекою для реалізації топологій обробки аудіо сигналу. За допомогою API низького рівня може бути ініційоване з’єднання з пристроєм відтворення або захоплення і аудіо потік може бути отриманий і відправлений до відповідного інтерфейсу. Модульна конструкція miniaudio дозволяє використовувати низькорівневий API без шкоди для використання інших функцій, таких як граф вузлів і менеджер ресурсів.

Система графів вузлів miniaudio надає користувачеві простий спосіб налаштувати розширені графіки мікшування та ефектів. Кожен вузол надсилає свій вихід на інший вузол, і так далі, щоб створити всі види ефектів. Зокрема, можуть бути реалізовані користувацькі блоки обробки та підключені відповідно до необхідної топології. Також підтримується

можливість маршрутизації та мікшування потоків даних [5].

3.1.3 Використання кросс-компіляції під час процесу розробки для вбудованих систем

Крос компіляція це процес компіляції програмного забезпечення на одній платформі (хост-системі) для виконання на іншій (таргет-системі). У вбудованих системах це необхідно, оскільки вбудовані пристрої зазвичай мають обмежені обчислювальні ресурси та не можуть виконувати компіляцію самостійно. Цей процес дозволяє значно скоротити час розробки вбудованої системи та виконати налагодження її основних компонентів на хост-платформі. Зазвичай процес кросс-компіляції включає до себе вибір цільової архітектури, налаштування крос-компілятора та додавання системи збирання до проекту з підтримкою крос-компіляції. У роботі використовується крос-компіляція в оточенні Docker для архітектури aarch64. Необхідні для збирання бібліотеки компілюються системою менеджменту пакетів conan для C++ [6].

У Лістингу 3.1 наведено реалізацію налаштувань кросс-компілятора для платформи Raspberry Pi 3B+ з архітектурою aarch64 і цільовою операційною системою Linux.

Лістинг 3.1 – Реалізація налаштувань крос-компілятора для системи збірки Cmake

```
# System information
set(CMAKE_SYSTEM_NAME "Linux")
set(CMAKE_SYSTEM_PROCESSOR "aarch64")
set(CROSS_GNU_TRIPLE "aarch64-rpi3-linux-gnu"
   CACHE STRING "The GNU triple of the toolchain to use")
set(CMAKE_LIBRARY_ARCHITECTURE aarch64-linux-gnu)
set(CPACK_DEBIAN_PACKAGE_ARCHITECTURE "arm64")

# Compiler flags
set(CMAKE_C_FLAGS_INIT       "-mcpu=cortex-a53+crc+simd")
set(CMAKE_CXX_FLAGS_INIT     "-mcpu=cortex-a53+crc+simd")
set(CMAKE_Fortran_FLAGS_INIT "-mcpu=cortex-a53+crc+simd")
```

Налагодження проекту виконується на хост-ОС з кінцевим збиранням фінального файлу виконання у середовищі Docker. Використання Docker дозволило значно скоротити час на розгортання середовища розробки та інструментів кросс-компіляції в оточенні MacOS. Зокрема, використання Docker дозволило виконати розгортання ізольованого середовища розробки з підтримкою Docker Containers Extension у VSCode.

Розгортання Docker у MacOS виконано шляхом віртуалізації та нативного розгортання контейнера для архітектури aarch64. За замовчуванням для процесорів M1 серії Docker виконує збирання образу системи для aarch64. У Лістингу 3.2 наведено фрагмент Dockerfile який використовується у процесі крос-компіляції додатку для платформи Raspberry Pi 3B+.

Лістинг 3.2 – Фрагмент Dockerfile для крос-компіляції проекту під платформу Raspberry Pi 3B+

```
FROM ubuntu
LABEL crossbuild-container
RUN apt-get update && \
    apt-get install -y \
    build-essential \
    bash \
    g++ \
    gcc \
    git \
    gzip \
    locales \
    python3 \
    file

RUN locale-gen en_US.utf8
RUN echo 'alias python=python3' > ~/.bashrc
RUN apt-get install -y pipx
RUN pipx ensurepath
RUN pipx install conan
RUN ln -s ~/.local/bin/conan /usr/bin/conan

WORKDIR /build_dir
RUN git clone https://github.com/tttapa/docker-arm-cross-toolchain.git
RUN conan remote add tttapa-docker-arm-cross-toolchain ./docker-arm-cross-toolchain
RUN apt-get install -y cmake
RUN conan profile detect
```

Для менеджменту бібліотек і залежностей у проекті використовується пакетний менеджер conan. Його використання включає до себе визначення залежностей, процедури генерації CMakeToolchainFile та створення відповідного CMakePresets.json який визначає можливі опції збирання проекту. З метою підтримки як збирання для хост оточення, так і збирання під цільову апаратну платформу було створено два conanfile.txt у якому визначено відповідні залежності як для запуску у headless версії Raspbian так і на хості з MacOS. Єдиною відмінністю є наявність SDL бібліотеки яка використовує апаратне прискорення GPU для малювання на відміну від devfb механізму, який використовується у headless запуску на Raspberry Pi.

Структура conanfile.txt для роботи у headless оточенні Raspberry Pi наведено у Лістингу 3.3.

Лістинг 3.3 – conanfile з декларацією бібліотек-залежностей проекту

```
[requires]
cxxopts/3.2.0
fmt/10.2.1
miniaudio/0.11.21
sigslot/1.2.2
atomic_queue/1.6.3
pffft/cci.20210511

[generators]
CMakeDeps
CMakeToolchain
[layout]
cmake_layout
```

Структура проекту для збирання розбита на декілька статичних бібліотек які задекларовані у CMakeLists.txt один фінальний виконуваний файл.

3.2 Використання інтерфейсу I2S

Інтерфейс I2S був розроблений у 1980-х роках, коли цифрові технології починали впроваджуватись на ринок споживчого аудіо. Його основною

метою є підтримка розвитку аудіоелектроніки через стандартизований інтерфейс для передачі цифрових даних між АЦП, ЦАП, цифровими фільтрами, процесорами цифрових сигналів та іншими мікросхемами, що використовуються в аудіосистемах. Це двоканальний протокол, спеціально створений для стереофонічного звуку. Загальну схему роботи інтерфейсу показано на рис. 3.1.

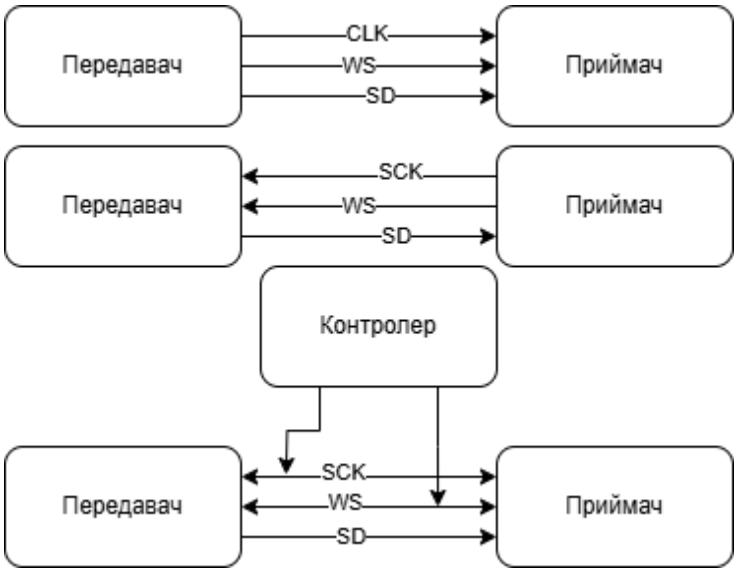

Рисунок 3.1 – Діаграма роботи протоколу Audio I2S

Дані передаються через лінію SD, при цьому стан лінії WS вказує на аудіоканал (лівий або правий), що передається в даний момент. Тактовий сигнал синхронізації надходить через лінію CLK. Як показано на рисунку 3.1, сигнали WS і SCK можуть генеруватися як передавачем, так і приймачем або іншим зовнішнім генератором [7].

Ось основні характеристики ліній послідовних даних протоколу:

– дані передаються у форматі MSb.

– передавач і приймач не зобов'язані мати узгоджену довжину слова: передавач відправляє те, що має, а приймач отримує те, що може обробити.

– нові біти даних можуть синхронізуватися за наростаючим або спадаючим фронтом тактового сигналу. Однак зазвичай синхронізація

відбувається по передньому фронту, тому більш простим є варіант, показаний на рис. 3.2, де дані задаються по задньому фронту, а синхронізуються по передньому фронту.

– протокол не передбачає невикористовуваних періодів синхронізації між словами: LSb одного слова безпосередньо слідує за MSb наступного слова.

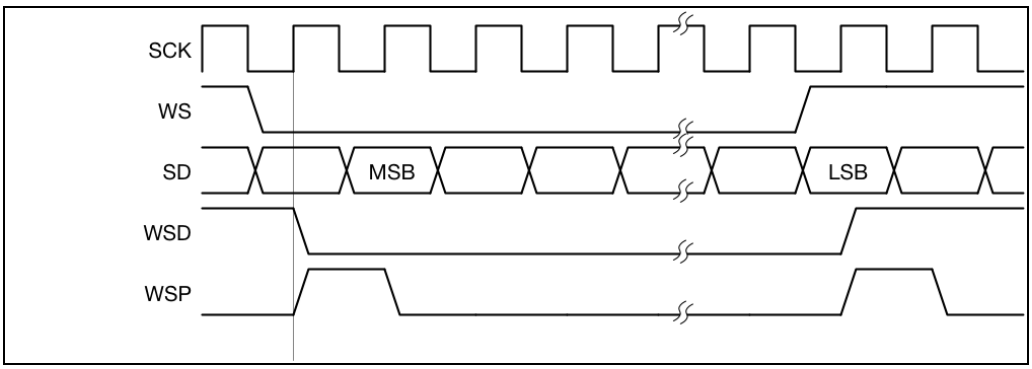

Рисунок 3.2 – Waveform роботи протоколу I^2S Audio

3.3 Опис алгоритмічних модулів системи

Алгоритмічна частина системи представлена компонентами рушія аудіо-ефектів, графічної підсистеми та механізму комунікації між аудіо підсистемою та графічним інтерфейсом. З метою запобігання проблем пов’язаних з артефактами під час обробки аудіо-потоку у реальному часі однією з головних задач було вирішення проблеми передачі даних з частини користувацького інтерфейсу до аудіо-ядра і передача результатів перетворення Фур’є до користувацького інтерфейсу.

Розглянемо реалізацію кожних з частин проекту більш детально.

Графічний інтерфейс було реалізовано з використанням бібліотеки LVGL яка є крос-платформний рішенням для вбудованих систем. Бібліотека є компактною та може бути розгорнута як на платформах з апаратним GPU, так і з використанням програмного рендерінгу інтерфейсу. LVGL містить у

собі велику кількість функціональних блоків користувацького інтерфейсу та можливість додавання вибіркових користувацьких обробників подій. Особливістю використання бібліотеки є ієрархічна структура графічних віджетів та можливість додавання метаданних об’єктів для подальшої реалізації обробки подій. З метою коректної реалізації ієрархії графічних компонентів всі основні функції системи розподілено по трьох контейнерних віджетах, до яких вже додаються елементи керування конкретними аудіо ефектами. Фрагмент створення контейнеру для обробки ефектів наведено у Лістингу 3.4.

Лістинг 3.4 - фрагмент створення корневого контейнеру для користувацького інтерфейсу з використанням бібліотеки LVGL

```
void add_effects_controls(lv_obj_t *parent)
    {
        lv_obj_t *grid = lv_obj_create(parent);
        lv_obj_set_size(grid, lv_pct(100), lv_pct(400)); // Fit to parent
        lv_obj_set_layout(grid, LV_LAYOUT_GRID);
        // Define grid template
        static lv_coord_t col_dsc[] = {LV_GRID_FR(1), LV_GRID_FR(1), LV_GRID_TEMPLATE_LAST};
        static lv_coord_t row_dsc[] = {LV_GRID_FR(1), LV_GRID_FR(1), LV_GRID_FR(1), LV_GRID_TEMPLATE_LAST};
        lv_obj_set_grid_dsc_array(grid, col_dsc, row_dsc);
        lv_obj_t *filter_control = lv_obj_create(grid);
        lv_obj_set_grid_cell(filter_control, LV_GRID_ALIGN_STRETCH, 0, 1, LV_GRID_ALIGN_STRETCH, 0, 1);
        add_flanger_controls(flanger_control);
        lv_obj_t *distortion_control_root = lv_obj_create(grid);
        lv_obj_set_grid_cell(distortion_control_root, LV_GRID_ALIGN_STRETCH, 0, 1, LV_GRID_ALIGN_STRETCH, 1, 2);
        add_distortion_controls(distortion_control_root);
        // lv_obj_t *master_volume_control = lv_obj_create(grid);
        // lv_obj_set_grid_cell(master_volume_control, LV_GRID_ALIGN_STRETCH, 0, 2, LV_GRID_ALIGN_STRETCH, 2, 1);
        // add_master_volume_controls(master_volume_control);
    }
```

У свою чергу, кожен елемент керування аудіо-ефектом представлений у вигляді зв’язки слайдеру, підпису який відповідає назві параметру що змінюється та активного підпису який відображає поточне значення параметру. Розглянемо детально реалізацію частини користувацького

інтерфейсу для зміни параметрів аудіо-ефекту Reverb.

Кожен блок керування аудіо-ефектом представлений у вигляді контейнера який містить у собі блоки користувацького інтерфейсу. Для цього блоку додаються базові елементи керування-слайдери, що містять мінімальне/максимальне значення параметру ефекту разом з кроком, що дозволяє змінювати значення ефекту. Також, кожен елемент користувацького інтерфейсу містить прив'язку до відповідного сигналу, який має бути ініційований під час зміни параметру. Відповідно, при зміні співвідношення зворотного зв'язку буде викликано onFeedback_Pct обробник події який передасть це значення до аудіо-рушія в окремому потоку виконання. Фрагмент реалізації ініціалізації слайдерів для керування ефектом реверберації наведено у Лістингу 3.5.

Лістинг 3.5 - Фрагмент ініціалізації слайдерів блоку керування ефектом реверберації з їх мінімальними і максимальними значеннями

```
// ====== Sliders for Parameters ======
        struct
        {
            const char *label;
            int min, max;
            int init;
            sigslot::signal<float> *propChangeEmmiter;
        } slider_params[] = {
            {"Wet Level (dB):", -24, 6, -3, &m_view_signals->reverb_signals.onwetLevel_dB},
            {"Dry Level (dB):", -24, 6, -3, &m_view_signals->reverb_signals.ondryLevel_dB},
            {"Feedback (%):", 0, 100, 0, &m_view_signals->reverb_signals.onfeedback_Pct},
            {"Left Delay (ms):", 0, 2000, 0, &m_view_signals->reverb_signals.onleftDelay_mSec},
            // {"Right Delay (ms):", 0, 2000, 0,&m_view_signals->reverb_signals.onrightDelay_mSec},
            {"Delay Ratio (%):", 0, 100, 100, &m_view_signals->reverb_signals.ondelayRatio_Pct}};
```

З метою реалізації можливості вмикання/вимикання аудіо-ефекту в аудіо-топології було реалізовано уніфікований bypass-режим для ефекту. Bypass є кнопкою, до якої прив'язаний ідентифікатор об'єкту аудіо ефекту. Під час обрання режиму bypass викликається загальна реалізація обробника

подій і пересилання ідентифікатора ефекту до аудіо-рушія який у свою чергу перемикає стан ефекта у bypass режим.

Фрагмент реалізації обробника події для bypass-режиму на стороні користувацького інтерфейсу наведено у Лістингу 3.6.

Лістинг 3.6 - Фрагмент обробки події обрання bypass-режиму з боку користувацького інтерфейсу

```
// Bypass button
        lv_obj_t *bypass_btn = lv_btn_create(parent);
        lv_obj_set_width(bypass_btn, lv_pct(50));
        lv_obj_add_flag(bypass_btn, LV_OBJ_FLAG_CHECKABLE);
        lv_obj_set_style_bg_color(bypass_btn, lv_palette_main(LV_PALETTE_GREY), 0);

        lv_obj_t *bypass_label = lv_label_create(bypass_btn);
        lv_label_set_text(bypass_label, "Bypass");
        lv_obj_center(bypass_label);

        lv_obj_add_event_cb(bypass_btn, bypass_button_click_event_handler,
LV_EVENT_ALL, object_identity.get());
```

Візуалізація перетворення Фур’є з метою отримання даних про поточну спектральну складову у сигналі була реалізована у вигляді окремого віджета LVGL який перевизначає процедуру відмальовування під час виклику бібліотеки та реалізує відображення як поточного спектру отриманого буферу семплів, так і відмальовування peak-hold червоними прямокутниками з метою відображення останнього максимального значення у заданому частотному діапазоні.

Віджет для малювання поточного спектру реалізований у вигляді контейнера який займає увесь доступний простір обмежений tabview.

У Лістингу 3.7 наведено фрагмент додавання віджету відображення спектру до основного користувацького інтерфейсу.

Лістинг 3.7 - Функція додавання віджету відображення спектру до LVGL Tabview контейнеру

```
void add_spectrum_view(lv_obj_t *parent)
{
    spectrum_obj = lv_obj_create(parent);
    lv_obj_remove_style_all(spectrum_obj);
    lv_obj_center(spectrum_obj);

    auto tab_height = lv_obj_get_height(parent);
    auto tab_width = lv_obj_get_width(parent);

    lv_obj_set_height(spectrum_obj, lv_obj_get_height(parent));
    lv_obj_set_width(spectrum_obj, lv_obj_get_width(parent));

    lv_obj_remove_flag(spectrum_obj, lv_obj_flag_t(LV_OBJ_FLAG_CLICKABLE | LV_OBJ_FLAG_SCROLLABLE));

    lv_obj_set_style_bg_color(spectrum_obj, lv_color_black(), 0);

    lv_obj_set_style_bg_opa(spectrum_obj, LV_OPA_100, 0);
    lv_obj_add_event_cb(spectrum_obj, &View::ViewImpl::spectrum_draw_event_cb, LV_EVENT_ALL,
                        this);
}
```

Спектральні біни відображаються шляхом відмальовування прямокутників які додатково масштабуються для відображення логарифмічних результатів у координатній сітці яка обмежена . Реалізацію відмальовування спектрограми наведено у Лістингу 3.8.

Лістинг 3.8 - Функція малювання спектральних бінів з використанням методів бібліотеки LVGL

```
void draw_spectrogram(lv_layer_t *layer, lv_area_t drawing_container_cords)
    {
        const auto ROOT_CONTAINER_WIDTH = (drawing_container_cords.x2 - drawing_container_cords.x1);
        const int32_t FREQ_BIN_WIDTH = ROOT_CONTAINER_WIDTH / (Common::Settings::FFT_SIZE / 2);

        for (int freq_bin_it = 0; freq_bin_it < Common::Settings::FFT_SIZE / 2; ++freq_bin_it)
```

```
            {
                lv_draw_rect_dsc_t freq_bin_rect_dsc{};
                lv_draw_rect_dsc_init(&freq_bin_rect_dsc);
                freq_bin_rect_dsc.bg_color = lv_color_white();
                freq_bin_rect_dsc.border_color = lv_color_black();
                freq_bin_rect_dsc.bg_opa = LV_OPA_COVER;
                freq_bin_rect_dsc.border_opa = LV_OPA_80;
                freq_bin_rect_dsc.border_width = 1;

                lv_area_t bin_coords = drawing_container_cords;
                bin_coords.x1 = bin_coords.x1 + freq_bin_it * FREQ_BIN_WIDTH;
                bin_coords.x2 = bin_coords.x1 + FREQ_BIN_WIDTH;
                bin_coords.y1 = map_to_log_scale(freq_domain_magnitudes[freq_bin_it], 0.0,
10.0, bin_coords.y2, bin_coords.y1);
                lv_draw_rect(layer, &freq_bin_rect_dsc, &bin_coords);
            }
        }
```

Перед відмальовуванням поточного спектру відбувається обчислення пікових значень та оновлення статусу peak-hold. Алгоритм роботи є наступним. Якщо значення магнітуди більше ніж зафіксоване попереднє значення - пікове значення миттєво оновлюється новим зафіксованим піком та скидається лічильник актуальності значення. Якщо нове пікове значення дорівнює останньому зафіксованому, то тільки скидається лічильник актуальності значення. Якщо нове пікове значення менше ніж останній зафіксований пік то відбувається інкремент покоління пікового значення що дозволяє анімувати плавний спад червоного прямокутника для peak-hold. Реалізація процедури оновлення та обчислення пікових значень спектральних бінів наведена у Лістингу 3.9.

Лістинг 3.9 - Реалізація функції обчислення пікових значень спектральних бінів

```
void update_spectrogram_bins()
    {
        const auto pComplexInterleaved = std::span<std::complex<float>>(reinterpret_cast<std::complex<float>*>(freq_domain_stored_samples_to_display.Read().first.data()), Common::Settings::FFT_SIZE / 2);

        for (int freq_complex_it = 0; freq_complex_it < Common::Settings::FFT_SIZE / 2; ++freq_complex_it)
        {
            auto complex_value = pComplexInterleaved[freq_complex_it];
            freq_domain_magnitudes[freq_complex_it] = std::abs(complex_value);
        }
        for (int freq_complex_it = 0; freq_complex_it < Common::Settings::FFT_SIZE / 2; ++freq_complex_it)
        {
            auto current_bin_magnitude = freq_domain_magnitudes[freq_complex_it];
            auto &peak_item = freq_domain_peaks[freq_complex_it];

        }
    }
```

З метою запобігання візуальних глітч-ефектів та переривчасті аудіо-потоку було реалізовано lock-free синхронізацію для відображення результатів перетворення Фур’є та передачі результатів обчислення з аудіо-потоку до потоку користувацького інтерфейсу. Реалізація була виконана на базі Triple buffering стратегії використання ресурсів та використання тільки атомарних примітивів синхронізації без задіювання системних механізмів синхронізації таких як std::mutex.

Фрагмент з імплементації triple buffering наведено у Лістингу 3.10.

Лістинг 3.10 - Фрагмент реалізації методу читання з контейнеру, що реалізує стратегію потрійної буферизації спектральних даних

```
std::pair<DataType&, bool>
    Read() {
        uintptr_t dirty_ptr = TRIO_VAR(middle_buffer_).load(kRelaxed);
        if ((dirty_ptr & kDirtyBit) == 0) {
            return {*front_buffer_, false};
        }
        uintptr_t prev =
TRIO_VAR(middle_buffer_).exchange(reinterpret_cast<uintptr_t>(front_buffer_), kAcqRel);
        front_buffer_ = reinterpret_cast<DataType*>(prev & kDirtyBitMask);
        return {*front_buffer_, true};
    }
```

У свою чергу, реалізація передачи кожної зміни параметру аудіо-ефекту реалізована з використанням lock-free черги, до якої передається структура, що містить у собі ідентифікатор ефекту, параметр що змінився та нове значення параметру. Параметри кожного ефекту містяться у відповідній структурі з використанням enum-class для визначення який з параметрів був змінений. Реалізацію структури що описує параметри аудіо-ефекту наведено у Лістингу 3.11

Лістинг 3.11 - Реалізація структури що описує параметри ефекту Distortion та відповідний enum class який визначає параметр що змінився

```
struct DistortionEffectParameterEvent
    {
        enum class TParameter
        {
            saturation, ///< saturation level
            asymmetry,  ///< asymmetry level
            outputGain, ///< outputGain level
```

```
            invertOutput, ///< invertOutput - triodes invert output
            enableHPF,    ///< HPF simulates DC blocking cap on output
            enableLSF,    ///< LSF simulates shelf due to cathode self biasing
            hpf_Fc,       ///< fc of DC blocking cap
            lsf_Fshelf,   ///< shelf fc from self bias cap
            lsf_BoostCut_dB, ///< boost/cut due to cathode self biasing
        };
        TParameter parameter;
        double parameter_value = 0.0;
    };
```

Для зберігання всього спектру можливих типів подій від ефектів використовується std::variant тип який є узагальненим контейнером для визначеної множини типів.

Слід зазначити, що вибір використання std::variant зумовлений необхідністю мінімізувати кількість виділення пам’яті в heap області.

З боку UI частини повідомлення про зміну параметра передається у чергу обробки подій, яка далі обробляється в аудіо потоку виконання.

Фрагмент передачі повідомлення від UI частини до real-time аудіо потоку наведено у Лістингу 3.12. Обробники вигляду post_onSomethingChanged є реалізацією типової моделі зв’язку об’єктів Publisher-Subscriber або реалізацією патерну проектування Observer.

Лістинг 3.12 - Фрагмент передачі повідомлення про зміну параметру Delay Ratio

```
void post_ondelayRatio_Pct(float value)
    {
        m_events_queue.push(
            Controller::Events::TEvent{
                Controller::Events::DelayEffectParameterEvent{.parameter = Controller::Events::DelayEffectParameterEvent::TParameter::delayRatio_Pct, .parameter_value = value}});
    }
```

З боку аудіо-рушія який виконується в окремому потоці виконання було додано обробник подій який періодично перевіряє чи є нові повідомлення від UI частини і обробляє їх відповідно до цільового блоку ефектів, до якого призначене повідомлення.

З метою кореткної та строго-типізованої обробки подій в аудіо-рушії було використано реалізацію патерну Visitor який може передати керування як спільним функціям блоку, так і перейти до встановлення конкретних параметрів в залежності від типу події.

Фрагмент реалізації обробника подій від різних блоків наведено у Лістингу 3.13.

Лістинг 3.13 - Реалізація обробки перенаправлення подій для встановлення параметрів різних блоків аудіо-ефектів

```
auto handler = overload{
        [this](const Controller::Events::OutputVolumeSetEvent &volume_event)
        {
            fmt::println(stdout, "Set volume level to:{}", volume_event.volume);
            m_volume_block->setVolume(volume_event.volume);
        },
        [this](const Controller::Events::ModulatedDelaySetEvent &modulated_effect_args)
        {
            m_modulated_effect->setParameters(modulated_effect_args);
        },
        [this](const Controller::Events::IirFilter &filter_block_event)
        {
            m_filter_iir->setFc(filter_block_event.parameter_value);
        },
        [this](const Controller::Events::DistortionEffectParameterEvent
&distortion_parameter_effect)
        {
            m_distortion_effect->setParameters(distortion_parameter_effect);
        },
```

Реалізація фільтру нижніх частот виконана на базі біквадратного БІХ фільтру якому можна встановити cutoff frequency в реальному часі.

Фрагмент оновлення коефіцієнтів фільтру в реальному часі наведено у Лістингу 3.14.

Лістинг 3.14 - Фрагмент розрахунку коефіцієнтів біквадратного фільтру в залежності від встановленої частоти зрізу

```
void calculateFilterCoeffs()
{
	double fc = zvaFilterParameters.fc;
	double Q = zvaFilterParameters.Q;
	vaFilterAlgorithm filterAlgorithm = zvaFilterParameters.filterAlgorithm;
	// --- normal Zavalishin SVF calculations here
	//     prewarp the cutoff- these are bilinear-transform filters
	double wd = kTwoPi*fc;
	double T = 1.0 / sampleRate;
	double wa = (2.0 / T)*tan(wd*T / 2.0);
	double g = wa*T / 2.0;
	// --- for 1st order filters:
	if (filterAlgorithm == vaFilterAlgorithm::kLPF1 ||
		filterAlgorithm == vaFilterAlgorithm::kHPF1 ||
		filterAlgorithm == vaFilterAlgorithm::kAPF1)
	{
		// --- calculate alpha
		alpha = g / (1.0 + g);
	}
}
```

3.4 Тестування розробленої системи створення аудіоефектів

Першопочаткове тестування системи було виконано у хост-оточенні з використанням інструментів налагодження LLDB та VSCode.Інтеграція на

фінальну апаратну платформу Raspberry Pi була виконана шляхом крос-компіляції фінального додатку та передачі зібраного бінарного виконуваного файлу з використанням rsync з метою мінімізації часу копіювання. Тестування системи було виконано шляхом програвання аудіо файлів і зміни параметрів ефектів з метою аудіо-візуального спостереження за поведінкою системи.

Першопочаткове тестування системи на хост-оточенні включає до себе перевірку коректності відображення елементів користувацького інтерфейсу та коректну обробку подій аудіо-підсистемою системи. Другий етап тестування у хост-оточенні включає до себе перевірку коректності реалізації алгоритмів обробки аудіо сигналу. На рис. 3.3 наведено результат запуску програмної частини системи з користувацьким інтерфейсом на платформі MacOS.

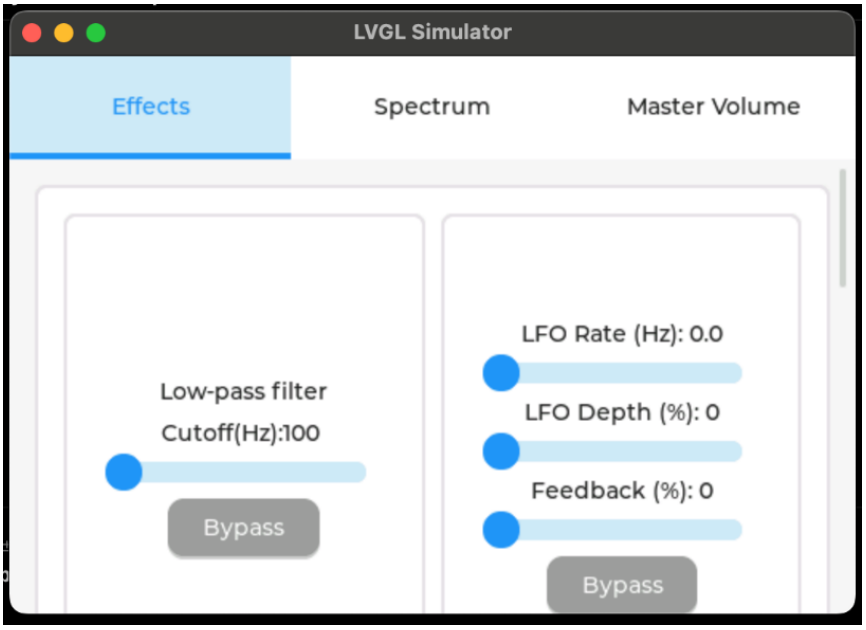

Рисунок 3.3 – Результат запуску системи у системі MacOS з метою тестування коректності відображення графічного інтерфейсу

Також у хост-оточенні була виконана перевірка коректності реалізації візуального відображення спектрограми в реальному часі. На рис. 3.4

зображено результат відображення спектрограми в реальному часі у запущеному додатку на системі MacOS.

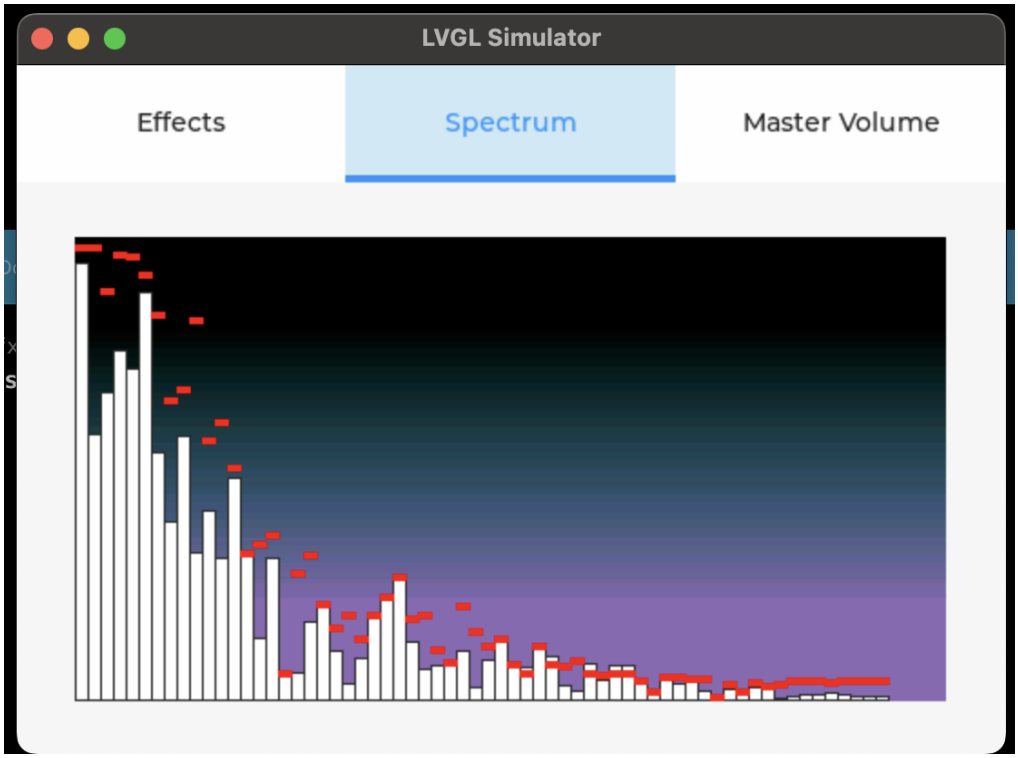

Рисунок 3.4 – Результат відображення фрагмента спектру аудіофайлу у реальному часі на MacOS

Для кожного з ефектів було реалізовано відповідний віджет який дозволяє змінювати параметри ефекту. На рис. 3.5 зображено слайдер керування частотою зрізу ФНЧ яка може бути змінена в реальному часі та кнопку Bypass яка дозволяє виключити ефект з ланцюгу обробки.

Переключення стану кнопки Bypass для всіх блоків ефектів супроводжується зміною її кольору на червоний, що визначає активацію Bypass-стану блоку.

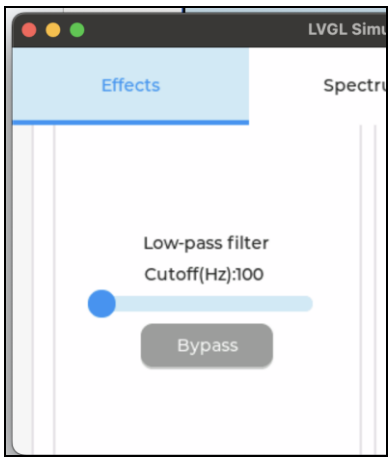

Рисунок 3.5 - Віджет керування частотою зрізу фільтру низьких частот та кнопка Bypass

Для блоку Flanger була реалізована підтримка зміни частоти LFO-осциллятора в реальному часі, зміни LFO стану та коефіцієнту зворотного зв'язку. Також, була додана підтримка загальної кнопки Bypass як і для інших ефектів. На рис. 3.6 наведено віджет який відповідає за налаштування ефекту Flanger.

Рисунок 3.6 - Віджет керування блоком Flanger

Параметри що були відображені для користувача у ефекті Reverb були значення wet/dry в децибелах, що відповідає насиченості симуляції об'ємної та маленької кімнати, можливість змінювати коефіцієнт зворотного зв'язку для попередніх відликів, значення затримки у мілісекундах та співвідношення затримки до вхідного сигналу під час їх змішування. Віджет, що призначений для керування блоком реверберації наведено на рис. 3.7.

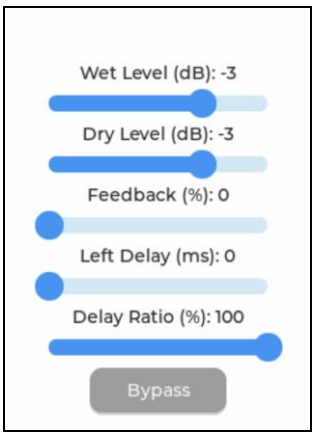

Рисунок 3.7 - Віджет керування блоком реверберації

Параметри, що доступні для ефекту Distortion наведені на рис. 3.8. Основними є насиченість, асиметричність та вихідний рівень підсилення на виході з блоку ефектів.

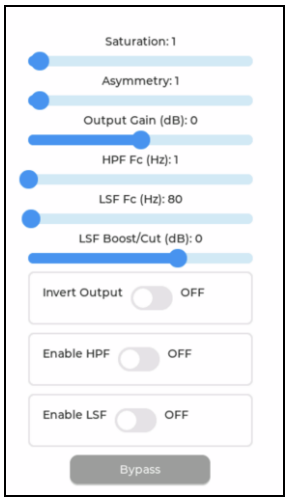

Рисунок 3.8 - Віджет керування блоком Distortion

З метою додавання можливості керування загальною гучністю на виході з системи було реалізовано слайдер майстер-гучності який має логарифмічну шкалу та за необхідності може повністю вимкнути аудіо-вихід шляхом натискання кнопки mute. Загальний вигляд віджету master volume наведено на рис. 3.9.

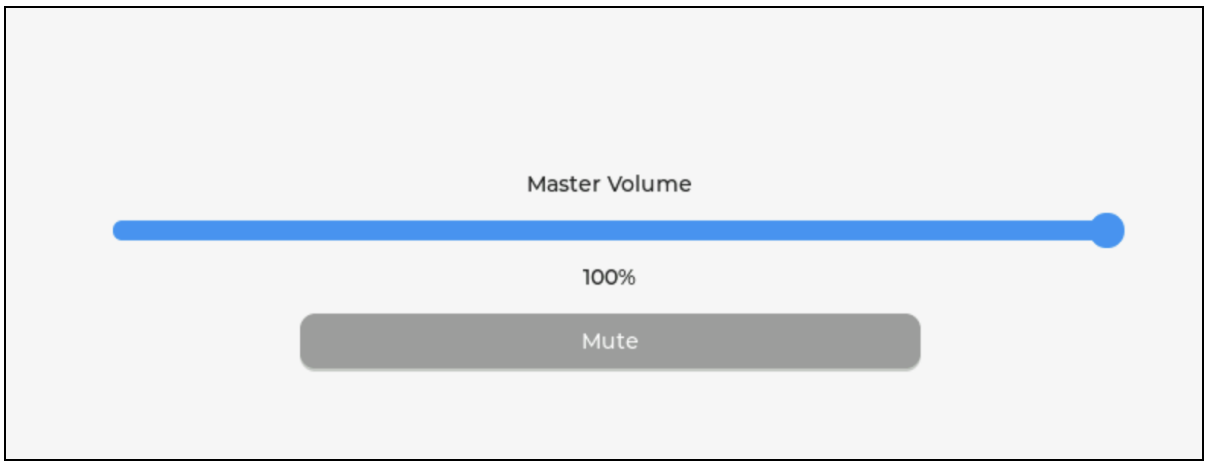

Рисунок 3.9 - Віджет керування загальною гучністю системи

ВИСНОВКИ

В ході виконання кваліфікаційної роботи розроблений прототип системи створення аудіо ефектів на платформі Raspberry Pi з використанням інструментальних засобів Docker, VSCode та мови програмування C++. Була виконана реалізація ефектів Reverb, Distortion та Flanger і IIR фільтру низьких частот.

Система використовує для відображення графічних даних TFT SPI дисплей. Відтворення сигналів на платі налагодження виконано за допомогою вбудованого I2S аудіо-кодеку.

Виконання макетування розробленої системи та проведено демонстраційний експеримент який підтвердив працездатність розброленого проекту. Параметри всіх ефектів відображаються на TFT дисплеї та можуть бути змінені через інтерфейс резестивного touch-screen-у.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Smith S. W. The scientist and engineer's guide to digital signal processing. 2nd ed. San Diego, Calif : California Technical Pub., 1999. 650 p.

2. Pirkle W. C. Nonlinear Processing: Distortion, Tube Simulation, and HF Exciters. Designing Audio Effect Plugins in C++. Second edition. | New York, NY : Routledge, 2019., 2019. P. 535–563.
URL: https://doi.org/10.4324/9780429490248-19 (date of access: 06.04.2025).

3. Robinson M. Getting Started with Juce. Packt Publishing, Limited, 2013.

4. Pommier J. Bitbucket. Bitbucket | Git solution for teams using Jira. URL: https://bitbucket.org/jpommier/pffft/src/master/ (date of access: 06.04.2025).

5. Reid D. Miniaudio project. https://miniaud.io/.

6. Шкіль О. Автоматизоване проєктування вбудованих систем цифрового оброблення сигналів на платформі SoC / О. Шкіль, Д. Рахліс, І. Філіпенко, В. Корнієнко, Т. Рожнова // Сучасний стан наукових досліджень та технологій в промисловості. – 2024. – № 1 (27). – С. 72-83.
DOI: https://doi.org/10.30837/ITSSI.2024.27.192.

7. Analysis of the implementation efficiency of digital signal processing systems on the technological platform SoC Zynq 7000 / Olexander Shkil, Oleh Filippenko, Dariia Rakhlis, Inna Filippenko, Valentyn Kornienko // Radioelectronic and Computer Systems. – 2024. – No. 4 (112). – P. 168-177. DOI: https://doi.org/10.32620/reks.2024.4.14