МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут автоматики і електротехніки
(повне найменування інституту, назва факультету)

Кафедра програмованої електроніки, електротехніки і телекомунікацій
(повна назва кафедри)

«Допущений до захисту»
Завідувач кафедри _____ В. М. Рябенький
д.т.н., професор
« 15 » грудня 2025 р.

Кваліфікаційна робота

на здобуття ступеня вищої освіти «магістр»
(освітньо-кваліфікаційний рівень)

на тему: Мікроконтролерна система цифрової обробки гітарного сигналу

Виконав: студент 6321М групи
_____ Мосійчук К.С.
(підпис)

Керівник роботи:
доцент каф. ПЕЕТ, к.т.н., доцент
(посада, науковий ступінь, вчене звання)
_____ О.С. Дьяконов
(підпис)

м. Миколаїв – 2025 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова
Навчально-науковий інститут автоматики та електротехніки

Кафедра програмованої електроніки, електротехніки і телекомунікацій
Спеціальність 171 «Електроніка»
Освітня програма Електронні системи

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ В. М. Рябенький

« 09 » жовтня 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «магістр»
(освітньо-кваліфікаційний рівень)

Студенту Мосійчуку Кирилу Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Мікроконтролерна система цифрової обробки гітарного сигналу

керівник роботи Дьяконов Олексій Сергійович, доцент каф. ПЕЕТ, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ректора № 1217 від «14» жовтня 2025 року.

2. Термін подання роботи: 09 грудня 2025 р.

3. Вихідні дані по роботі: Огляд технологій цифрової обробки аудіосигналів. Дослідження принципів роботи гітарних ефектів.

4. Перелік питань, що належать до розробки (найменування розділів) Розробка мікроконтролерного гітарного процесора на базі STM32 з можливістю оцифрування, цифрової обробки та відтворення аудіосигналу. Принципи цифрової обробки аудіосигналів у реальному часі. Огляд та аналіз гітарних ефектів. Розробка структурної та принципіальної схеми пристрою. Розробка макету та програмного забезпечення для нього.

5. Перелік презентаційних матеріалів Структурна схема пристрою. Принципіальна схема пристрою. Схема живлення пристрою. Схема підключення мікроконтролеру. Схема вхідного фільтру. Схема вихідного фільтру. Схема підключення периферії. Блок-схеми алгоритмів ефектів. Фото осцилограм. Інтерфейс меню.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Тубальцев А.М.		

7. Дата видачі завдання «15» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Огляд цифрових методів обробки аудіосигналів та існуючих гітарних процесорів	Вересень 2025 р.	виконано
2	Формування технічних вимог до мікроконтролерного гітарного процесора	Жовтень 2025 р.	виконано
3	Розробка апаратної частини пристрою	Листопад 2025 р.	виконано
4	Розробка програмної частини пристрою	Листопад 2025 р.	виконано
5	Охорона праці	Грудень 2025 р.	виконано
6	Оформлення роботи	Грудень 2025 р.	виконано

Студент _____ Мосійчук К.С.
(підпис) (прізвище та ініціали)

Керівник роботи _____ О.С. Дьяконов
(підпис) (прізвище та ініціали)

Анотація

В рамках цієї магістерської роботи був розроблений гітарний процесор на базі мікроконтролера STM32F446RE, призначений для цифрової обробки аудіосигналу в реальному часі. У роботі виконано проєктування апаратної частини пристрою, зокрема вхідного фільтру, системи живлення та вихідного фільтру. Реалізовано програмне забезпечення з використанням периферії STM32: АЦП, ЦАП, DMA, таймерів. Здійснено програмну реалізацію основних гітарних ефектів, таких як overdrive, delay, chorus, tremolo, еквалайзеру, а також допоміжних фільтрів. Додатково розроблено інтерфейс користувача на базі дисплея SSD1306 і системи керування кнопками.

Ключові слова: гітарний процесор, цифрова обробка сигналів, STM32, АЦП, ЦАП.

Abstract

In this master thesis, a guitar processor for digital audio processing which is based on the STM32F446RE microcontroller, was developed. The work includes the design of the hardware part of the device, in particular the input filter, power supply system and output filter. The software is implemented using STM32 peripherals: ADC, DAC, DMA, timers. The software implementation of the main guitar effects, such as overdrive, delay, chorus, tremolo, equalizer, as well as auxiliary filters, has been done. Additionally, a user interface based on the SSD1306 display and button control system has been developed.

Keywords: guitar processor, digital signal processing, STM32, ADC, DAC.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

ADC – Analog-to-Digital converter

DAC – Digital-to-Analog converter

DMA – Direct Memory Access

FIR – Finite Impulse Response

IIR – Infinite Impulse Response

ФВЧ – Фільтр високих частот

ФНЧ – Фільтр низьких частот

ВСТУП

Сучасна індустрія музичного обладнання активно рухається у напрямку цифрової обробки аудіосигналів. Те, що раніше вимагало складних аналогових схем з операційними підсилювачами, діодними ланцюгами, фільтрами та індуктивностями, сьогодні може бути реалізовано у вигляді компактних цифрових пристроїв на базі мікроконтролерів. Така тенденція зумовлена розвитком мікроелектроніки, появою потужних 32-бітних мікроконтролерів та доступністю алгоритмів цифрової обробки сигналів (DSP), які дозволяють отримати звучання, близьке до класичних аналогових ефектів. Саме тому розробка цифрових гітарних процесорів стала окремим напрямом у сфері аудіотехніки та вимагає поєднання знань з програмування, схемотехніки та теорії сигналів.

У даній роботі виконано розробку гітарного процесора на базі мікроконтролера STM32F446RE, здатного в реальному часі здійснювати захоплення, обробку та відтворення аудіосигналу. Мікроконтролери сімейства STM32F4 мають архітектуру Cortex-M4 із вбудованим DSP-модулем, що забезпечує апаратну підтримку операцій множення-накопичення, необхідних для реалізації фільтрів, модульованих ефектів та інших алгоритмів обробки. Це дозволяє реалізувати ефекти, які традиційно створювалися складними аналоговими схемами: overdrive, chorus, delay, tremolo, фільтрацію.

Особливістю цієї роботи є не лише розробка алгоритмів DSP, а й побудова повноцінної системи, що включає апаратну та програмну частину. Розроблений пристрій підтримує високошвидкісний захват сигналу за допомогою вбудованого АЦП з використанням DMA, містить аналогові фільтри, а також виводить параметри ефектів на OLED-дисплеї на базі контролера SSD1306, які можна змінювати за допомогою панелі управління.

Під час створення цифрових ефектів було досліджено методи моделювання аналогових схем, розробка цифрових фільтрів, принципи реалізації нелінійних спотворень.

Практична частина роботи охоплює розробку апаратної схеми, вибір компонентів, побудову вхідних та вихідних каскадів, програмну реалізацію всіх ефектів, оптимізацію продуктивності.

Метою даної роботи є створення компактного, функціонального та дешевого гітарного процесора, здатного відтворювати основні ефекти обробки сигналу та забезпечувати стабільну роботу.

					171.6321М.КР.ПЗ.Р1	
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*		
Розробник		Мосійчук К.С.			Передумови застосування цифрової обробки сигналів у гітарних ефектах	*Літ.* / *Арк.* 9 / *Аркушів*
Консультант						
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова
Керівник		Дьяконов О.С.				
Зав. каф.		Рябенький В. М.				

РОЗДІЛ 1. ПЕРЕДУМОВИ ЗАСТОСУВАННЯ ЦИФРОВОЇ ОБРОБКИ СИГНАЛІВ У ГІТАРНИХ ЕФЕКТАХ

1.1. Актуальність роботи

Об’єктом цього дослідження є процеси цифрової обробки аудіосигналу з електогітари у реальному часі на основі мікроконтролерів.

Предметом дослідження є алгоритми реалізації гітарних ефектів (overdrive, delay, tremolo, chorus, та фільтрація сигналу) у вбудованих системах з обмеженими ресурсами.

У сучасній музиці все частіше використовують саме гітарні процесори через їхню більшу гнучкість та точність у відтворенні аналогових ефектів, і з розвитком цифрових технологій різниця між цифровим та аналоговим звучанням становиться все меншою. Такі процесори є і більш компактними, ніж стандартні аналогові pedalboards, які складаються з багатьох педалей та проводів, що робить використання та пересування усього цього менш зручною.

Але професійні комерційні рішення є дорогими, що стає перепоною для багатьох любителей цього музичного інструменту. Моєю метою є побудова саме дешевої та гнучкої платформи, яка буде доступна для кожного.

1.2. Електрогітари

Електрогітара – це гітара, яка, на відміну від класичної або акустичної, використовує для підсилення гучності свого звучання не форму корпусу, а зовнішній електричний підсилювач. Цей інструмент використовує звукознімачі для перетворення вібрації струни у магнітному полі звукознімача на електричний сигнал, який потім підсилюється та знову перетворюється на звук у динаміку.

Електрогітара була розроблена Джорджем Бошамом у 1932 році, а у 1950-1960 рр. стала найпопулярнішим інструментом у поп-музиці. Цей

музичний інструмент використовується у багатьох жанрах: рок, кантрі, метал, джаз, блюз та багато інших.

1.3. Поява педалей гітарних ефектів

Різні аудіо-ефекти стали невід'ємною частиною гри на цьому музичному інструменті. Мабуть, найпершим з ефектів з'явився ефект distortion, або ж overdrive. Цей ефект з'являвся тоді, коли на перших гітарних підсилювачах використовували гучність більшу, ніж це було заплановано при їх розробці, або ж при їх пошкодженні.

У 1962 році компанія Maestro випустила легендарну педаль FZ-1 Fuzz-Tone, що стала першим серійним гітарним ефектом.

У 1970-ті роки розвиток транзисторних схем дозволив створити педалі таких ефектів як phaser, flanger та chorus.

У цих же роках з'явилися перші педалі з ефектом reverb, які на той час використовували пружини, а також аналогові педалі з ефектом delay на базі мікросхем BBD (Bucket Brigade Device), які зберігають та передають сигнал у вигляді заряду між послідовно з'єднаними конденсаторами.

З розвитком цифрових технологій, ефекти почали реалізовуватися завдяки цифровій обробці сигналів, що дало можливість використовувати їх за меншу ціну, зробило педалі більш компактними, та дало можливість комбінувати ці ефекти у одному процесорі ефектів.

1.4. Використання цифрової обробки сигналів

Цифрова обробка сигналів має певні переваги над аналоговими методами. При цифровій обробці, є можливість легко змінювати параметри системи, без зміни апаратної частини. Цифрова система менше залежить від точності компонентів та не деградує з часом. А також завдяки цифровій обробці можливо реалізовувати більш складні алгоритми.

Цифрова обробка можлива завдяки АЦП та ЦАП, які дають можливість зчитувати аналоговий сигнал, перетворювати його у набір числових значень, які потім проходять обробку, та знову перетворювати їх у аналоговий сигнал.

Частота дискретизації за теоремою Котельникова, частота дискретизації сигналу, тобто частота АЦП, має бути вдвічі більшою за частоту самого сигналу, щоб уникнути аліасингу (накладання високочастотної складової сигналу на низькочастотну, що призводить до спотворення сигналу). Стандартними частотами дискретизації аудіосигналу є частоти 44.1 кГц та 48 кГц.

При використанні АЦП неминучим є так званий шум квантування. Оскільки цифрова область складається лише з цифрових слів скінченної довжини, які повинні представляти безперервний сигнал, крок перетворення вносить похибку квантування.

Щоб зменшити його вплив, можна використовувати такий алгоритм як oversampling. У такому випадку, дискретизація сигналу виконується с частотами у 2*N разів більшою ніж частота сигналу (N – ступінь двійки). Потім виконується фільтрування (наприклад, використання фільтру ковзного середнього) та децимація (зменшення частоти дискретизації). Oversampling зменшує шум квантування, аліасинг, та може збільшити точність АЦП. Наприклад, дискретизація з частотою у 16 разів більшою ніж частота вхідного сигналу дає ще один значущий біт АЦП, збільшуючі його точність.

					171.6321М.КР.ПЗ.Р2	
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*		
Розробник		Мосійчук К.С.			Апаратна частина	*Літ.* / *Арк.* 13 / *Аркушів*
Консультант						
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова
Керівник		Дьяконов О.С.				
Зав. каф.		Рябенький В. М.				

РОЗДІЛ 2. АПАРАТНА ЧАСТИНА

2.1. Вимоги до пристрою

Перед цим пристроєм поставлені такі вимоги:

- Пристрій повинен забезпечувати безперервне приймання, обробку та виведення гітарного сигналу із загальною затримкою не більше 10–15 мс, що є критичним для комфортної гри.
- Пристрій повинен реалізовувати базові фільтри для поліпшення якості звучання та пригнічення шумів.
- Має бути присутня взаємодія з користувачем, який повинен мати можливість змінювати параметри ефектів

2.2. Вибір компонентів та розробка макету

Пристрій буде складатися з таких компонентів:

- Мікроконтролер
- Вхідний роз’єм Jack 6.35 мм, вихідний роз’єм Jack 3.5 мм
- Дисплей SSD1306
- Операційні підсилювачі NE5532
- Кнопки, потенціометр
- Резистори та конденсатори

Мікроконтролер:

У цій роботі використовується мікроконтролер STM32F446RE, зображення якого наведено на рис. 2.1. Вибір мікроконтролера зумовлений наступними вимогами:

- Мікроконтролер потребує високу частоту роботи, щоби мати більшу кількість тактів на кожну вибірку сигналу, який поступає з частотою 48 кГц. Тобто, мікроконтролер з частотою 48 МГц буде мати лише 1000 тактів на вибірку, коли мікроконтролер з частотою 180 МГц – 3750.

- Повинен мати прискорення математичних операцій, таких як: цілочисельне множення, цілочисельне ділення, побітове зсування та тригонометричні операції, що пришвидшить виконання обробки сигналу.

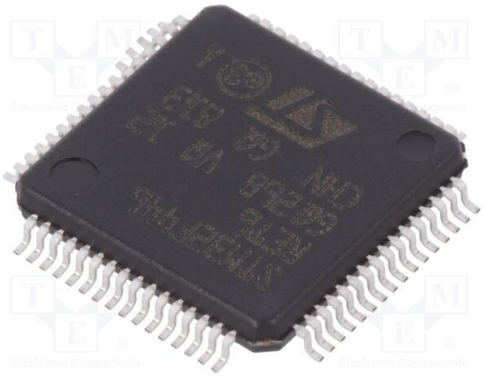

Рисунок 2.1 – Мікроконтролер STM32F446RE

Цей мікроконтролер має такі характеристики:

- 32-розрядний процесор ARM® Cortex®-M4 із FPU
- Максимальна частота процесора 180 МГц
- Напруга живлення від 1,7 В до 3,6 В
- 512 КБ Flash
- Системний SRAM 128 КБ
- Backup SRAM 4 КБ
- Таймери загального призначення (10)
- Таймери Advanced-Control (2)
- Основні таймери (2)
- SPI (4)
- I2S (2)
- USART (4)
- UART (2)
- Повношвидкісний і високошвидкісний USB OTG
- CAN (2)
- SAI (2)

- SPDIF-Rx (1)
- HDMI-CEC (1)
- Quad SPI (1)
- GPIO (50) із можливістю зовнішнього переривання
- 12-розрядний АЦП (3) з 16 каналами
- 12-розрядний ЦАП з 2 каналами

Для макету буде використовуватися плата Nucleo. Її вигляд та розпіновка наведені на рис 2.2, рис. 2.3, рис. 2.4.

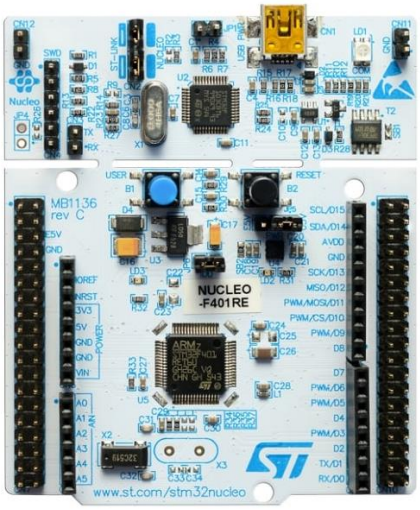

Рисунок 2.2 – Плата NUCLEO-F446RE

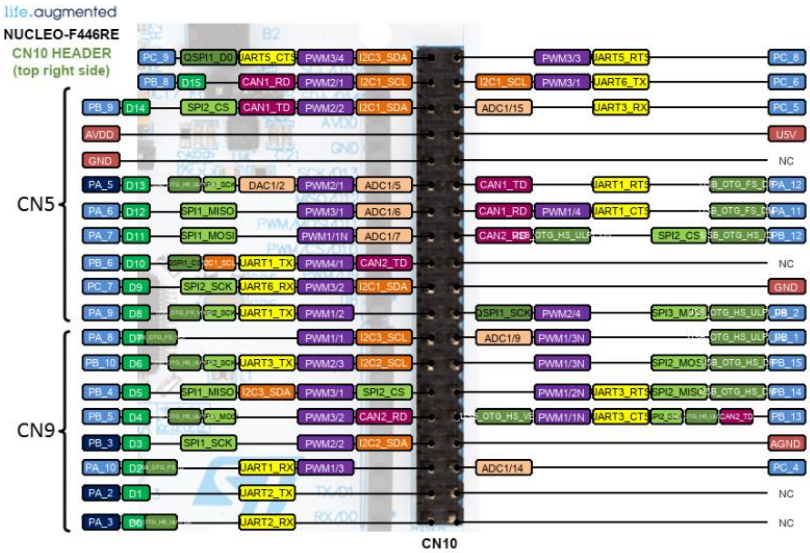

Рисунок 2.3 – Піни плати NUCLEO-F446RE

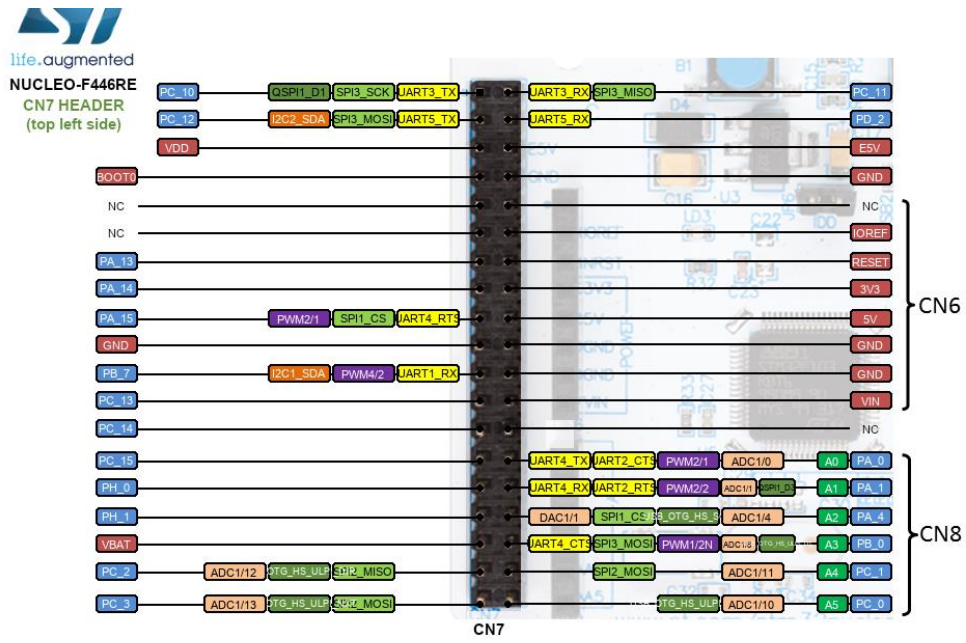

Рисунок 2.4 – Піни плати NUCLEO-F446RE

В якості дисплея використовується SSD1306, зовнішній вигляд якого наведено на рис 2.5.

Рисунок 2.5 – Дисплей SSD1306

Комунікація з цим дисплеєм виконується за інтерфейсом I2C. Максимальна частота для цього інтерфейса в мікроконтролері STM32F446RE – 400 кГц, у режимі Fast Mode. Існує також можливість працювати у режимі FMPI2C (Fast-mode Plus) з частотою 1 МГц. Цей дисплей має роздільну здатність 128x64 пікселя, і використовується саме через маленьку кількість пікселей, які потрібно оновлювати.

Для побудови аналогових фільтрів та буферів використовується операційний підсилювач NE5532 зовнішній вигляд та розпіновка якого наведені на рис 2.6.

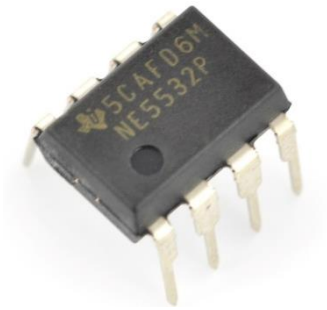

Рисунок 2.6 – Операційний підсилювач NE5532

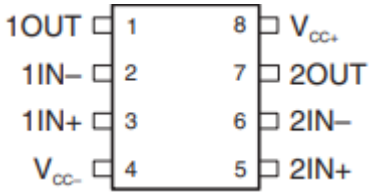

Рисунок 2.7 – піни операційного підсилювача NE5532

Ці операційні підсилювачі використовуються через їх гарні характеристики, такі як невеликий шум – лише 5 нВ√Гц при 1 кГц, велика смуга пропускання при коефіцієнті підсилювання 1 – 10 МГц, великий коефіцієнт ослаблення синфазного сигналу – 100 дБ, та добру швидкість зміни вихідного сигналу – 9 В/мкс.

Структурна схема пристрою наведена на рис 2.8.

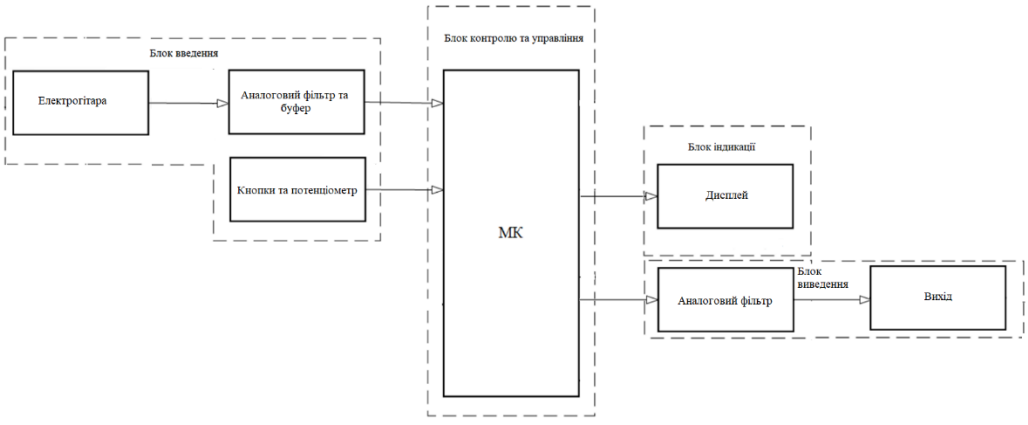

Рисунок 2.8 – Структурна схема пристрою

Фотографія макету пристрою наведена на рис. 2.9

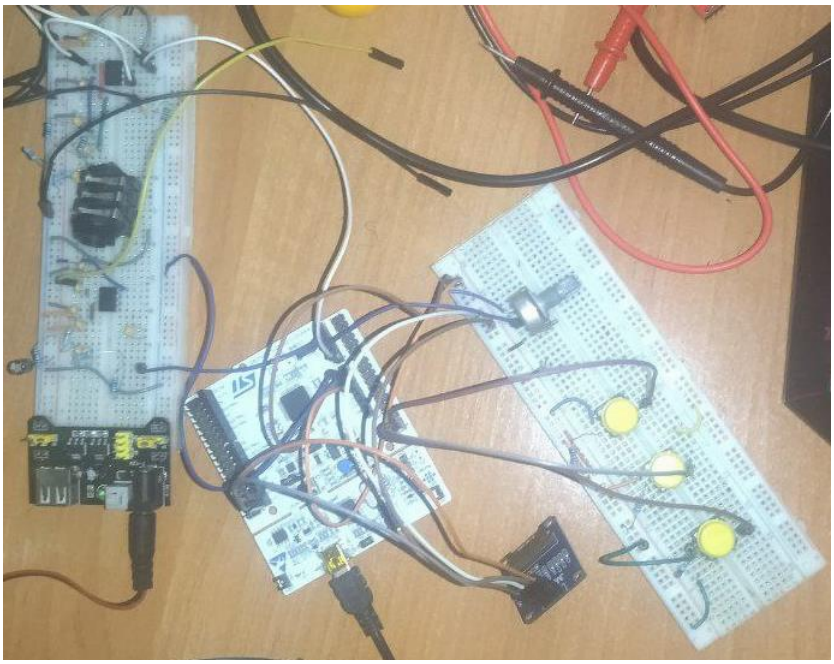

Рисунок 2.9 – Макет пристрою

2.3. Розробка пристрою

Принципіальна схема пристрою розробляється за допомогою програмного забезпечення KiCad та наведена на рис 2.10.

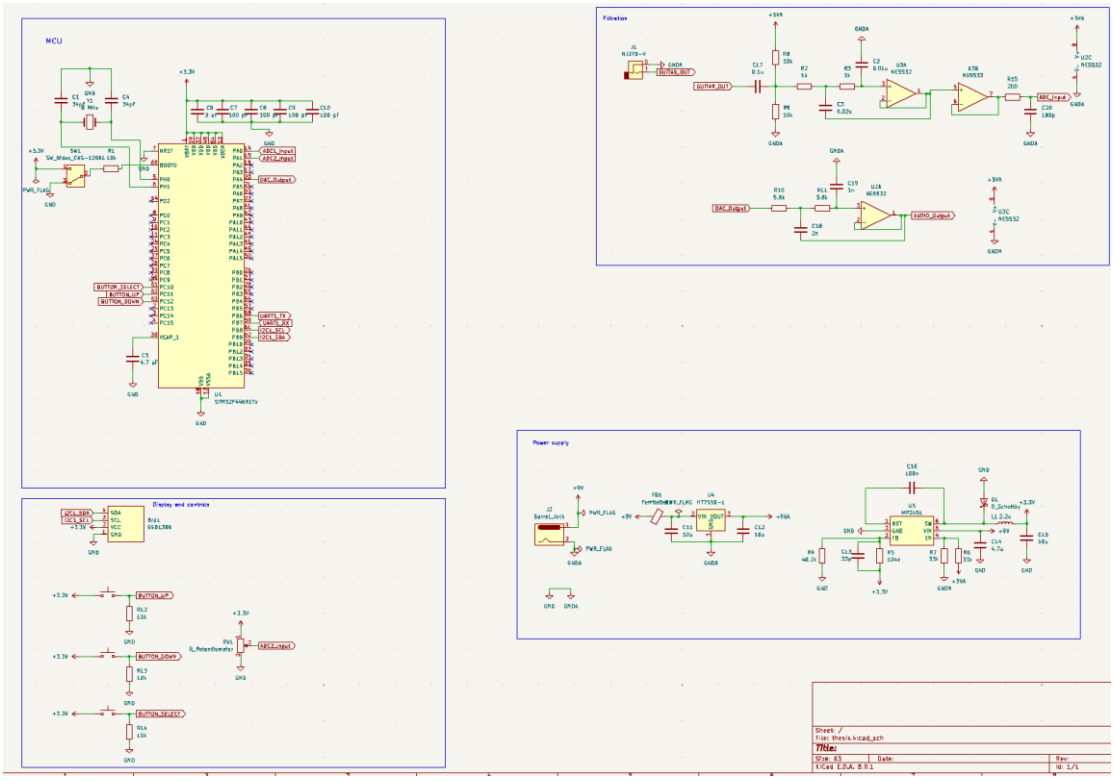

Рисунок 2.10 – Принципіальна схема пристрою

Пристрій буде живитися через роз’єм DC Barrel, який є стандартним для гітарних педалей. Блоки живлення для педалей, один з яких наведений на рис. 2.11, мають на виході 9 В

Рисунок 2.11 – Блок живлення для педалі ефектів

Але 9 В це забагато як для аналогової, так і для цифрової частини пристрою. Тому потрібно цю напругу знизити. Схема живлення пристрою наведена на рис. 2.12.

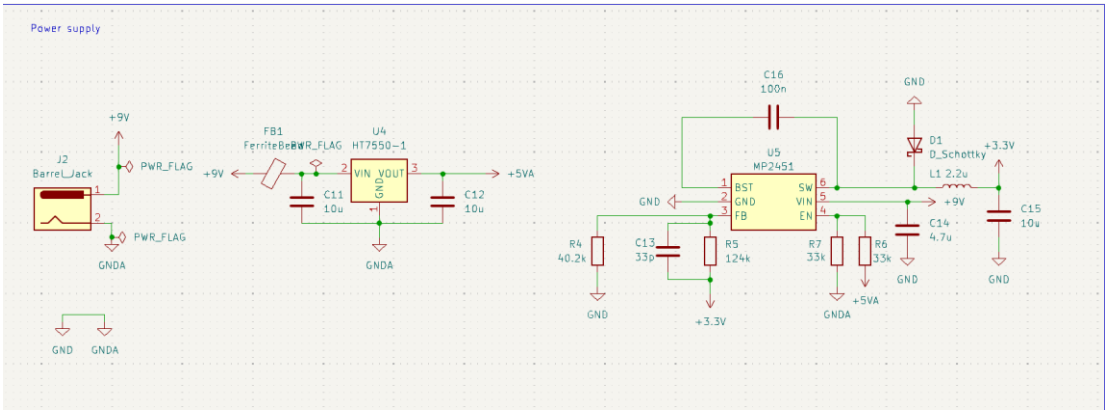

Рисунок 2.12 – Схема живлення пристрою

Аналогова та цифрова частина живляться від різних джерел. Для живлення аналогової частини використовується лінійний перетворювач HT7550-1 від компанії HOLTEK. Лінійні перетворювачі (LDO), хоч і менш ефективні ніж понижуючі перетворювачі напруги (buck converters) та більше

нагріваються, у розробці аудіотехніки виявляють свій головний плюс, а саме – маленьку кількість шумів.

Принцип роботи LDO полягає в тому, що він вимірює вихідну напругу через резистивний дільник напруги, який масштабує вихідний рівень. Масштабована вихідна напруга подається на підсилювач помилки, де вона порівнюється з опорною напругою. Підсилювач помилки керує послідовним прохідним приладом для підтримки потрібної напруги на вихідному терміналі. Тобто, не виконується ніяких перемикань, як це робиться у імпульсних перетворювачах. Для розуміння роботи цього перетворювача на рис. 2.13 наведена його блок-діаграма.

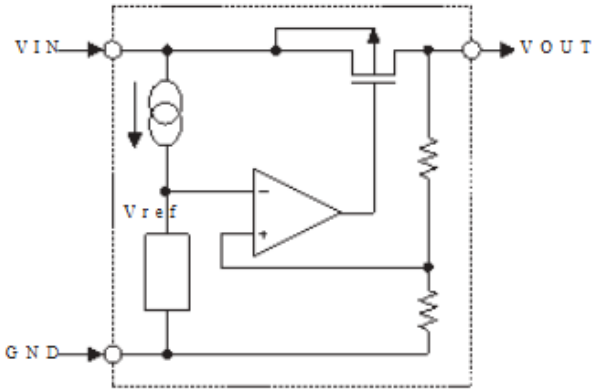

Рисунок 2.13 – Блок-діаграма мікросхеми HT7550-1

Схема підключення цієї мікросхеми наведена на рис. 2.14.

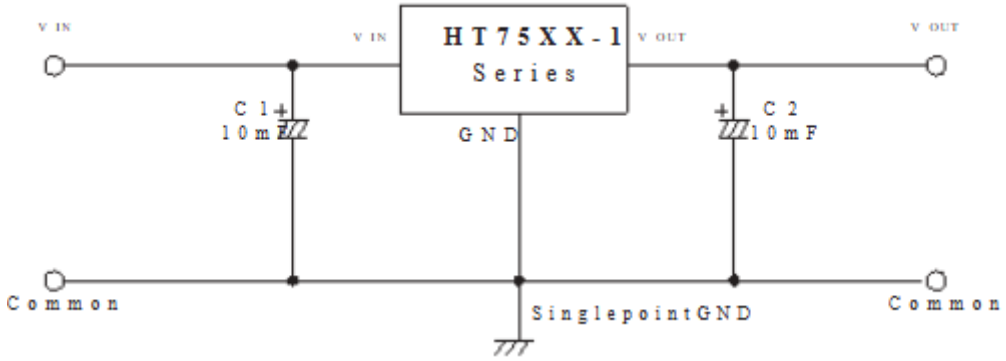

Рисунок 2.14 – Схема підключення мікросхеми HT7550-1

Для живлення аналогової частини використовується напруга 5 В, щоб запобігти кліппінгу при менших значеннях напруги.

Для живлення цифрової частини пристрою використовується напруга 3.3 В, яка досягається за допомогою перетворювача напруги MP2451.

MP2451 — це високочастотний (2 МГц) імпульсний понижувальний перетворювач напруги з інтегрованим внутрішнім високовольтним силовим MOSFET-транзистором. Його вхідна напруга від 3.3 до 36 В, а вихідна напруга регулюється від 0.8 В до 0.8*Vin В. Його типова схема підключення зображена на рис. 2.15.

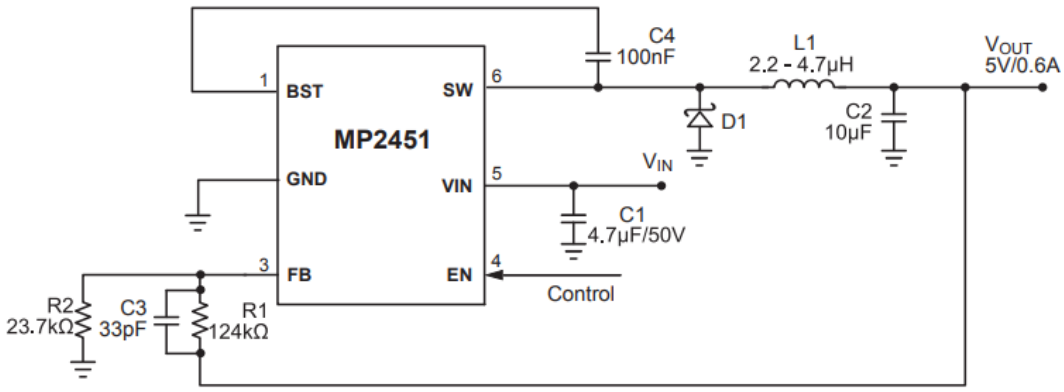

Рисунок 2.15 – Схема підключення мікросхеми MP2451

Вихідна напруга регулюється резисторами R1 та R2. R1 приймається за 124 кОм, а R2 розраховується за формулою:

$$R2 = \frac{R1}{\frac{Vout}{0.8V} - 1}$$

Індуктивність L1 розраховується за формулою

$$L1 = \frac{Vout}{fs * \Delta I_L} * (1 - \frac{Vout}{Vin})$$

Де fs – частота перемикання, ΔI_L – значення від піка до піка пульсуючого струму індуктивності.

В якості фільтрів, вхідного та вихідного, використовуються фільтри Баттервота 2-го порядку за топологією Салена-Кі, а саме у конфігурації з

одиничним коефіцієнтом підсилення. Вхідний фільтр та буфер зображені на рис 2.16.

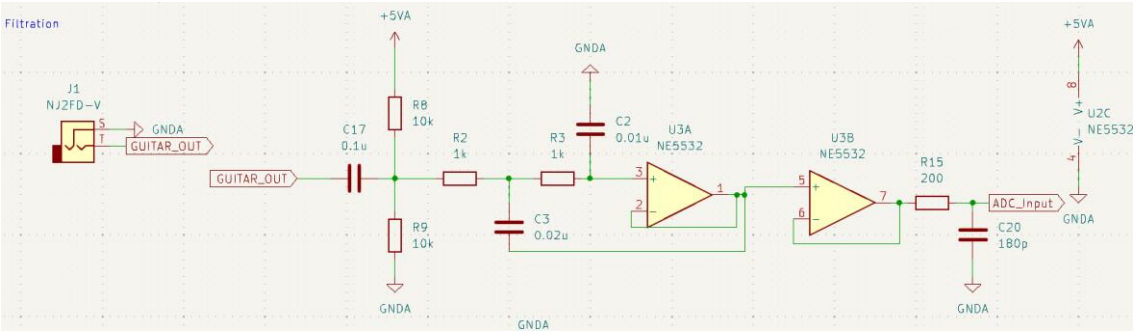

Рисунок 2.16 – Вхідний фільтр та буфер

У цій конфігурації, частота зрізу визначається формулою:

$$f_0 = \frac{1}{2\pi R\sqrt{C_1 C_2}}$$

А добротність фільтра визначається формулою:

$$Q = \frac{1}{2}\sqrt{\frac{C_1}{C_2}}$$

Тобто, якщо значення ємності C_2 в два рази менше за C_1, то добротність буде дорівнювати 0.707, що і є добротністю фільтра Баттервота.

Було проведено моделювання цих фільтрів у Multisim, і були отримані наступні результати, що зображені на рис. 2.17.

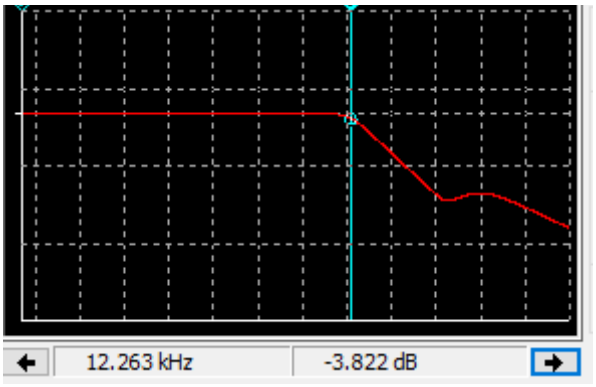

Рисунок 2.17 – АЧХ вхідного фільтра

Для вхідного фільтра я обрав частоту зрізу 12 кГц, тому що, хоча при стандартному налаштуванні електрогітари найвища частота дорівнює приблизно 1.2 кГц, але присутні також і гармоніки більших частот, які треба зберегти для отримання більш живого звуку.

Після вхідного фільтру також йде буфер – повторювач на операційному підсилювачі та ФНЧ. Він потрібен для того, щоб зменшити вихідний опір фільтра, та дати можливість конденсатору всередині АЦП мікроконтролера зарядитись.

Вихідний фільтр був встановлений на виході АЦП з метою зменшити аліасинг та шуми квантизації. Його схема та АЧХ наведені на рис. 2.18 та рис. 2.19.

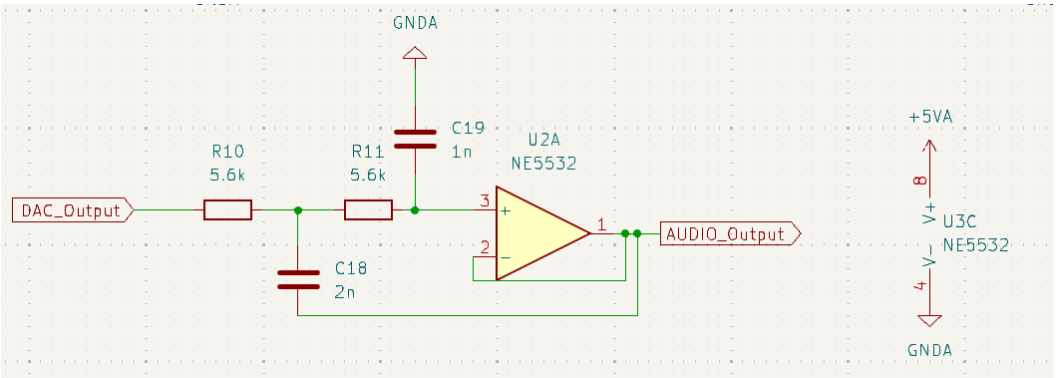

Рисунок 2.18 – Вихідний фільтр

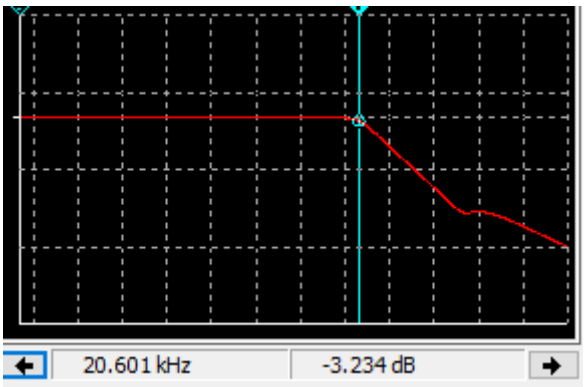

Рисунок 2.19 – АЧХ вихідного фільтра

У цьому фільтрі частота зрізу становить 20 кГц.

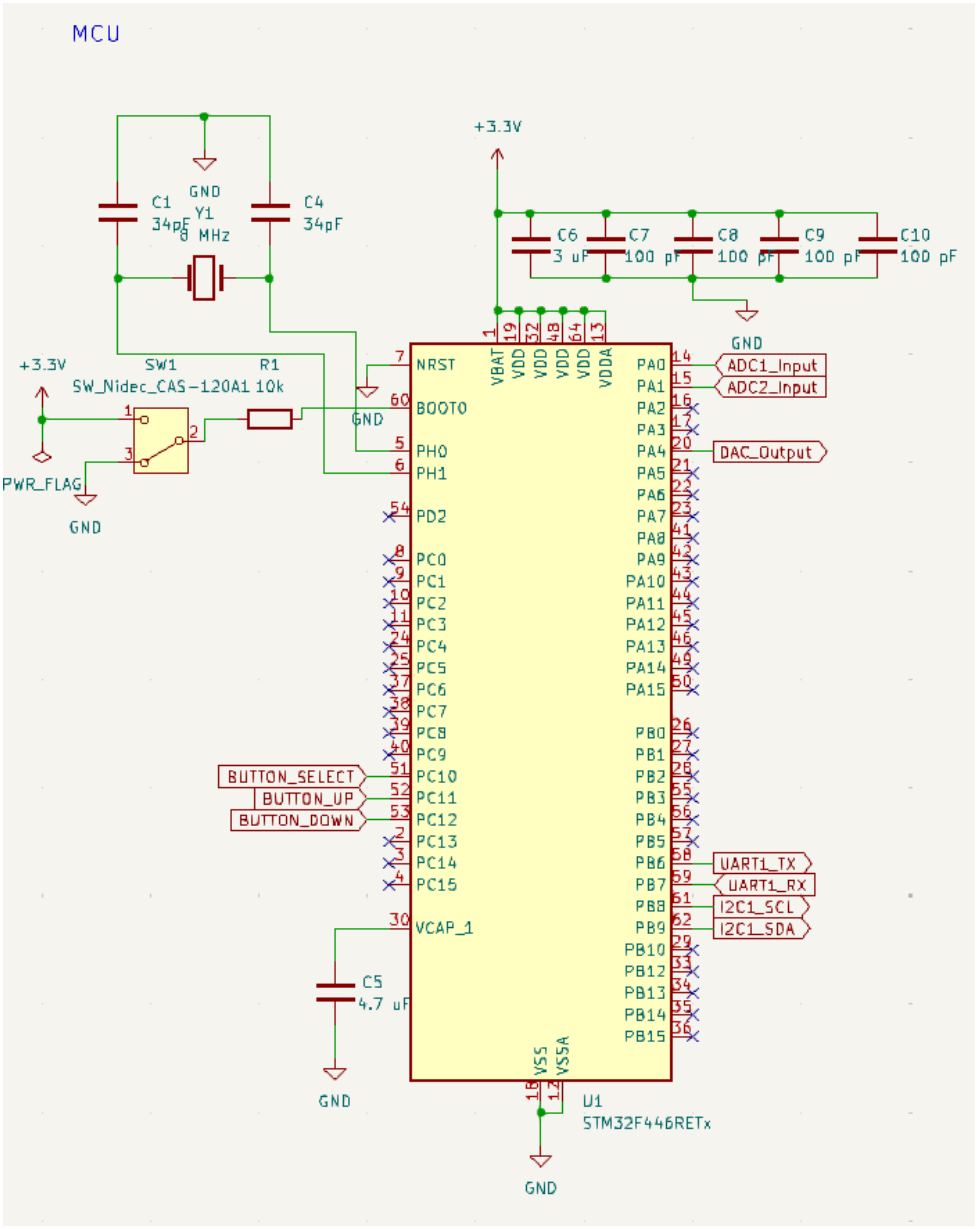

Рисунок 2.15 – Схема підключення мікроконтролеру STM32F446RE

До самого мікроконтролеру, схема підключення якого наведена на рис 2.15, підключений кварцовий генератор – HSE з частотою 8 МГц, що відповідає за тактування мікроконтролера. До входів живлення підключені розв’язуючі конденсатори, їхні значення обрані за документацією на мікроконтролер. До піну VCAP_1 також підключений конденсатор, що відповідає за стабілізацію вбудованого регулятору напруги.

Схема підключення елементів керування та дисплею наведені на рис. 2.16.

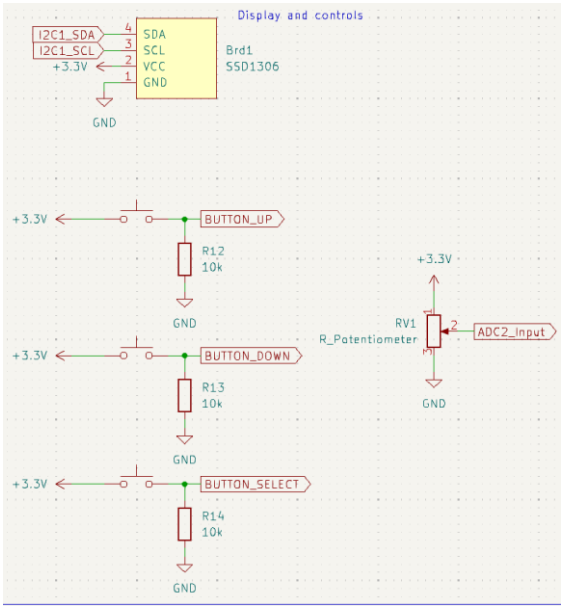

Рисунок 2.16 – Підключення дисплею та елементів керування

Дисплей працює за допомогою інтерфейса I2C, живиться від 3.3 В. Кнопки мають pull-down резистори, коли вони не натиснуті – на їх виході 0 В, при натисканні – 3.3 В.

Розпіновка мікроконтролера STM32F446RE наведена на рис. 2.16.

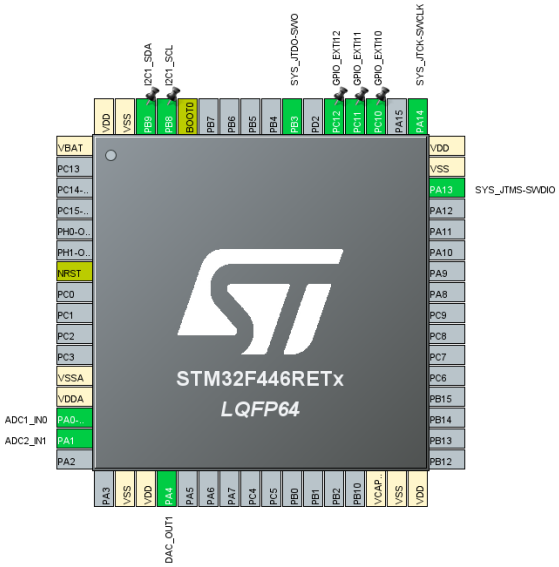

Рисунок 2.16 – Розпіновка мікроконтролера STM32F446RE, зеленим кольором позначені задіяні піни

Налаштування тактування пристрою наведено на рис. 2.17.

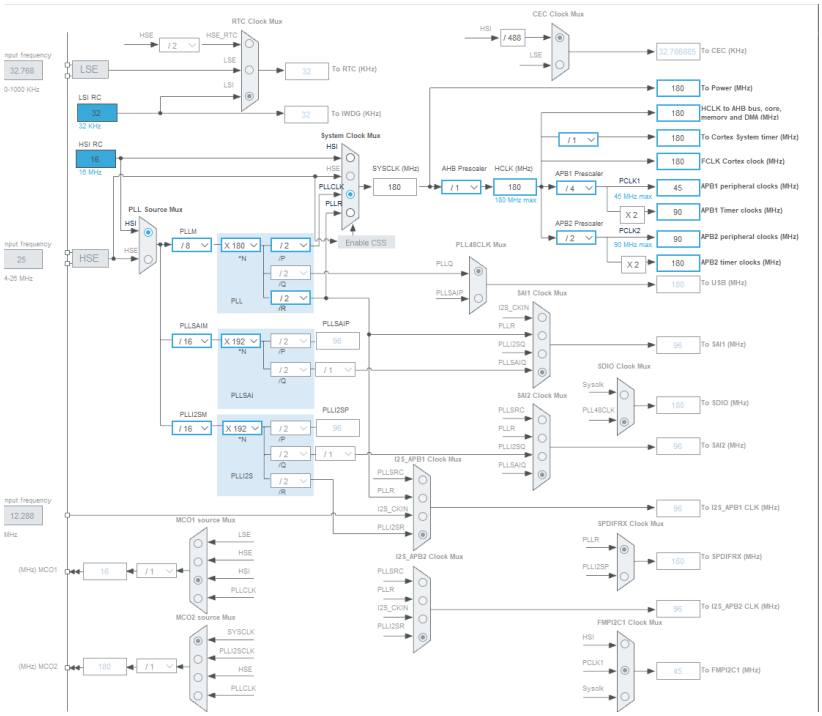

Рисунок 2.17 – Налаштування тактування пристрою

2.4. Висновки до другого розділу

У цьому розділі була спроектована апаратна частини гітарного процесора на базі мікроконтролера STM32F446RE. Були сформовані критерії вибору компонентів та побудована структурна та принципіальні схеми пристрою.

Було реалізовано вхідний аналоговий тракт, який включає буферизацію та первинне формування сигналу за допомогою операційних підсилювачів NE5532, що забезпечують достатній запас по швидкості наростання, низькі шумові параметри та стабільну роботу в аудіодіапазоні. Архітектура аналогової частини передбачає поділ живлення на дві лінії: 5 В для аналогових каскадів та 3.3 В для цифрових модулів, що дозволило зменшити рівень взаємних завад та стабілізувати роботу АЦП мікроконтролера.

Для отримання стабільних напруг живлення була використана комбінація:

– buck-перетворювач MP2451 для живлення цифрової частини;

– LDO-регулятор для аналогової частини, який забезпечує низький рівень шумів на виході.

Такий підхід дозволив мінімізувати пульсації напруги та покращити загальну якість аудіосигналу.

Було проєктовано схему підключення мікроконтролера STM32F446RE, яка включає АЦП для оцифрування аудіосигналу, OLED-дисплей на контролері SSD1306 та модуль керування кнопками.

					171.6321М.КР.ПЗ.РЗ		
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*			
Розробник		Мосійчук К.С.			Програмна частина	*Літ.*	*Арк.* 29 / *Аркушів*
Консультант							
Н. контр.		Добрінова Л. П.				НУК імені адмірала Макарова	
Керівник		Дьяконов О.С.					
Зав. каф.		Рябенький В. М.					

РОЗДІЛ 3. ПРОГРАМНА ЧАСТИНА

3.1. Налаштування мікроконтролера

Потрібно налаштувати ADC1, ADC2, ЦАП, Timer2, Timer3, I2C1, та GPIO.

Конфігурація ADC1 наведена на рис 3.1.

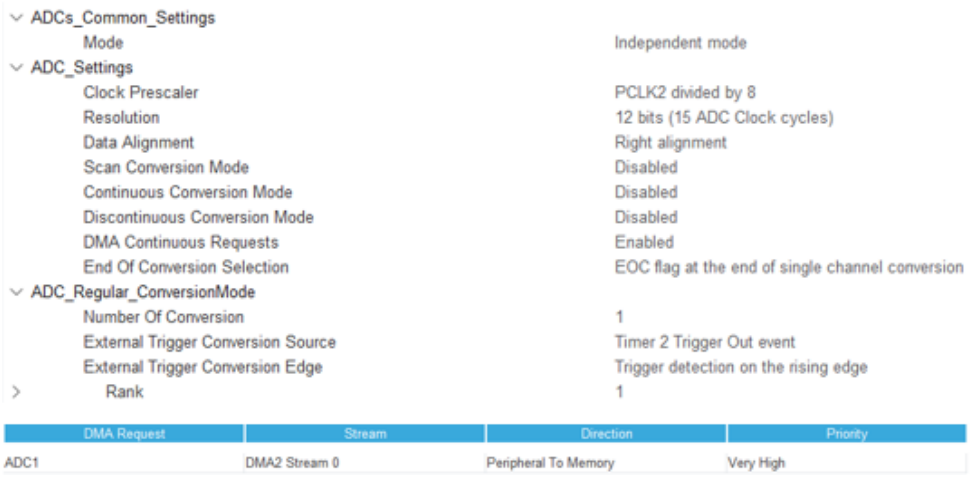

Рисунок 3.1 – Конфігурація ADC1

Цей АЦП працює у режимі DMA, що дає можливість передавати інформацію між пам’яттю та периферією без участі процесора, і це підвищує швидкість роботи тим чином, що у процесора буде більше семплів саме на обробку сигналу. Застосовується DMA Continuous Request з тією метою, щоб DMA не зупиняв роботу після обробки першого пакету даних. Цей АЦП викликається по перериванню від Timer 2.

Для виводу сигналу використовується ЦАП, що також викликається за перериванням Timer2 та використовує DMA. Його конфігурація наведена на рис 3.2.

Рисунок 3.2 – Конфігурація DAC

Також треба зазначити, що пріоритети DMA виставлені як «дуже високі», з тією метою, щоб ніяка інша задача не переривала та не затримувала прийом та передачу сигналу. Також, DMA працює з шириною інформації Half Word, тобто 16 байт.

Конфігурація Timer 2 наведена на рис 3.3.

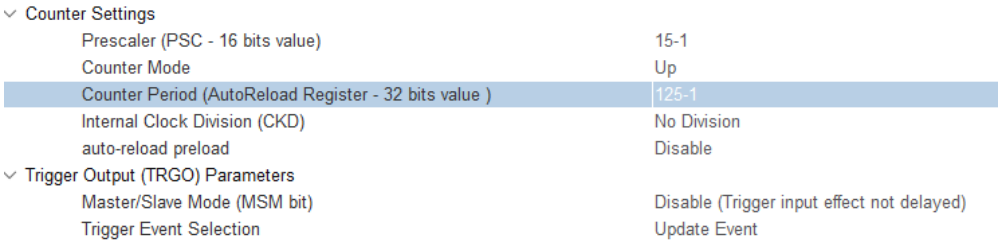

Рисунок 3.3 – Конфігурація Timer2

Цей таймер тактується від внутрішнього годинника APB1, з частотою 90 МГц. Щоб отримати частоту таймера 48 кГц, потрібну для роботи АЦП, в регістри TIM2_ARR (Auto-Reload Register) та TIM2_CNT (Counter) виставляються потрібні значення.

ADC2 конфігурується таким же чином, як і ADC1, за тим вийнятком, що викликається за перериванням Timer3. Цей АЦП використовується для зчитування напруги з потенціометра. Timer3 працює з частотою 10 Гц, з тією ж метою як можливо менше впливати на обробку самого сигналу.

I2C1 працює у режимі Fast Mode, що дає йому можливість працювати з частотою 400 кГц, а не 100 кГц.

Піни PC10, PC11, PC12 налаштовані у режим External Interrupt, тобто зміна логічного рівня на них викликає переривання. PC11 та PC12 викликають переривання по фронту, а PC10 викликає і про фронту, і по зрізу.

3.2. Ініціалізація.

Відбувається ініціалізація ADC1, ADC2, DMA, GPIO, Timer2, Timer3, I2C1. Ініціалізується дисплей SSD1306.

Створюються структури фільтрів еквалайзеру та ефектів хорус та тремоло.

```
Biquad lowshelf;
Biquad highshelf;
Biquad peakingEQ;
Tremolo tremolo;
Chorus chorus;
```

Ініціалізується фільтр високих частот

```
HPF_coefficient = 1.0f/(1.0f+alpha);
HPF_output[0] = 0; HPF_output[1] = 0;
HPF_input[0] = 0; HPF_input[1] = 0;
```

Розраховуються початкові значення для фільтрів еквалайзеру

```
lowshelf_calculation(&lowshelf, 45000, 200, 1, 0);
highshelf_calculation(&highshelf, 45000, 4000, 1, 0);
peak_calculation(&peakingEQ, 45000, 750, 0.7, 0);
```

А також ініціалізуються delay та FIR фільтр низьких частот.

```
delay_init(100);
filter_init();
```

3.3. Програмна реалізація еквалайзера

Задля реалізації фільтрів, що використовуються у еквалайзері, застосовуються біквадратні фільтри, які є фільтрами IIR, тобто з нескінченною імпульсною характеристикою.

Структурний граф такого фільтру наведений на рис 3.4.

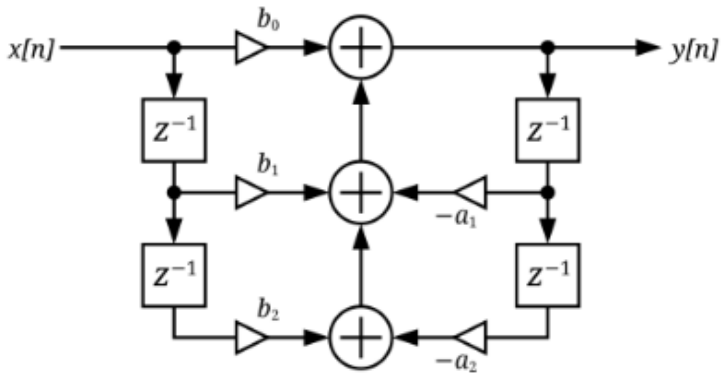

Рисунок 3.4 – Структурний граф біквадратного фільтра

Їхня передаточна функція виглядає таким чином:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$$

Зазвичай виконується нормалізація, і $a_0 = 1$. У такому випадку, передаточна функція буде виглядати так:

$$H(z) = \frac{\frac{b_0}{a_0} + \frac{b_1}{a_0} z^{-1} + \frac{b_2}{a_0} z^{-2}}{1 + \frac{a_1}{a_0} z^{-1} + \frac{a_2}{a_0} z^{-2}}$$

З цього виходить таке рівняння:

$$y[n] = \frac{b_0}{a_0} x[n] + \frac{b_1}{a_0} x[n-1] + \frac{b_2}{a_0} x[n-2] - \frac{a_1}{a_0} y[n-1] - \frac{a_2}{a_0} y[n-2]$$

Де x – вхідні семпли, а y – вихідні.

Структура біквадратного фільтра виглядає таким чином:

```
typedef struct {
    float b0, b1, b2;
    float a1, a2;
    float x1, x2;
    float y1, y2;
} Biquad;
```

Ініціалізація фільтрів виконується так, що попередні вхідні та вихідні семпли прирівнюються до нуля, а також відбувається нормалізація коефіцієнтів:

```
void biquad_filter_init(Biquad* filter, float b0, float b1, float b2,
float a0, float a1, float a2)
{
      filter->b0 = b0/a0;
```

```
        filter->b1 = b1/a0;
        filter->b2 = b2/a0;
        filter->a1 = a1/a0;
        filter->a2 = a2/a0;
        filter->x1 = 0;
        filter->x2 = 0;
        filter->y1 = 0;
        filter->y2 = 0;
    }
```

Для підсилення певних частот я використовую три типи фільтрів: low-shelf для підсилення низьких частот, peak для підсилення середніх та high-shelf для підсилення високих.

АЧХ та ФЧХ цих фільтрів наведені на рис. 3.5, рис 3.6, рис. 3.7.

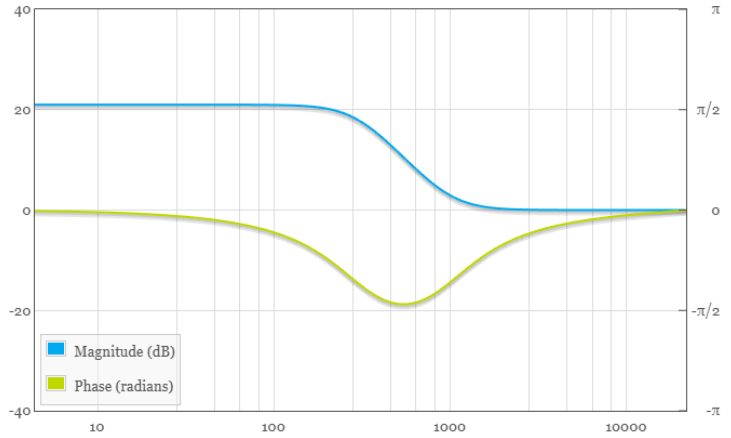

Рисунок 3.5 – АЧХ та ФЧХ low-shelf фільтра

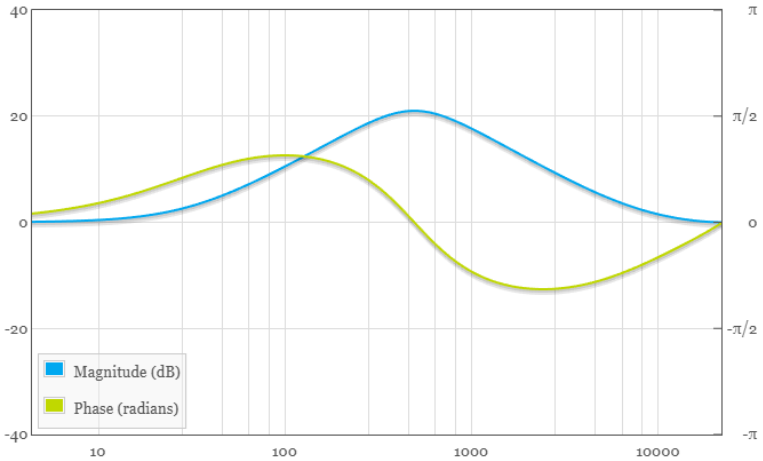

Рисунок 3.6 – АЧХ та ФЧХ peak фільтру

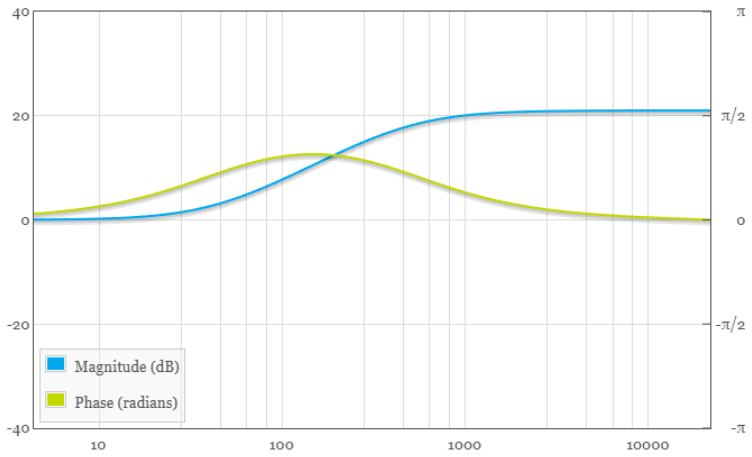

Рисунок 3.7 – АЧХ та ФЧХ high-shelf фільтра

Коефієнти для цих фільтрів розраховуються за формулами, наданими Робертом Брістоу-Джонсом у його «Audio EQ Cookbook».

Функція для low-shelf фільтра:

```
void lowshelf_calculation(Biquad* filter, int fs, int f0, float s, int
dBgain)
{
	float A = pow(10, (float)dBgain/40);
	float w0 = 2*M_PI*f0/fs;
	float cos_w0 = cos(w0);
	float sin_w0 = sin(w0);
	float alpha = sin_w0/2*sqrt((A+1/A)*(1/s-1)+2);

	float b0 = A*((A+1)-(A-1)*cos_w0+2*sqrt(A)*alpha);
	float b1 = 2*A*((A-1)-(A+1)*cos_w0);
	float b2 = A*((A+1)-(A-1)*cos_w0-2*sqrt(A)*alpha);
	float a0 = (A+1)+(A-1)*cos_w0+2*sqrt(A)*alpha;
	float a1 = -2*((A-1)+(A+1)*cos_w0);
	float a2 = (A+1)+(A-1)*cos_w0-2*sqrt(A)*alpha;
	biquad_filter_init(filter, b0, b1, b2, a0, a1, a2);
}
```

Функція для peak фільтра:

```
void peak_calculation(Biquad* filter, int fs, int f0, float Q, int
dBgain)
{
	float A = pow(10, (float)dBgain/40);
	float w0 = 2*M_PI*f0/fs;
	float cos_w0 = cos(w0);
	float sin_w0 = sin(w0);
	float alpha = sin_w0/(2*Q);

	float b0 = 1+alpha*A;
	float b1 = -2*cos_w0;
```

```
        float b2 = 1-alpha*A;
        float a0 = 1+alpha/A;
        float a1 = -2*cos_w0;
        float a2 = 1-(alpha/A);
        biquad_filter_init(filter, b0, b1, b2, a0, a1, a2);

}
```

Функція для high-shelf фільтра:

```
void highshelf_calculation(Biquad* filter, int fs, int f0, float s, int
dBgain)
{
        float A = pow(10, (float)dBgain/40);
        float w0 = 2*M_PI*f0/fs;
        float cos_w0 = cos(w0);
        float sin_w0 = sin(w0);
        float alpha = sin_w0/2*sqrt((A+1/A)*(1/s-1)+2);

        float b0 = A*((A+1)+(A-1)*cos_w0+2*sqrt(A)*alpha);
        float b1 = -2*A*((A-1)+(A+1)*cos_w0);
        float b2 = A*((A+1)+(A-1)*cos_w0-2*sqrt(A)*alpha);
        float a0 = (A+1)-(A-1)*cos_w0+2*sqrt(A)*alpha;
        float a1 = 2*((A-1)-(A+1)*cos_w0);
        float a2 = (A+1)-(A-1)*cos_w0-2*sqrt(A)*alpha;
        biquad_filter_init(filter, b0, b1, b2, a0, a1, a2);
}
```

Функція для розрахунку вихідного значення семпла:

```
float biquad_calculation(Biquad* filter, float x)
{
        float y = filter->b0*x+filter->b1*filter->x1+filter->b2*filter-
>x2-filter->a1*filter->y1-filter->a2*filter->y2;
        filter->y2 = filter->y1;
        filter->y1 = y;
        filter->x2 = filter->x1;
        filter->x1 = x;
        return y;
}
```

3.4. Програмна реалізація ефекту tremolo

Ефект tremolo використовує амплітудну модуляцію сигналу. Для цієї модуляції використовується сигнал певної форми та частоти, що менша за частоту модулюємого сигналу.

Принцип роботи та блок-схема цього гітарного ефекту наведені на рис. 3.8 та рис. 3.9.

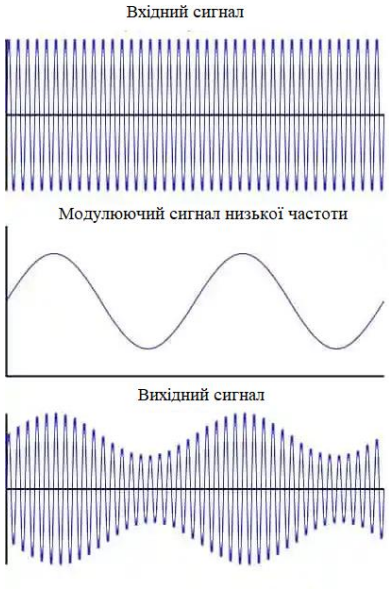

Рисунок 3.8 – Принцип роботи tremolo

Для модуляції може використовуватися синусоїда або сигнал пилкоподібної форми. У цій роботі для збереження швидкодії системи я буду використовувати саме пилкодібний сигнал.

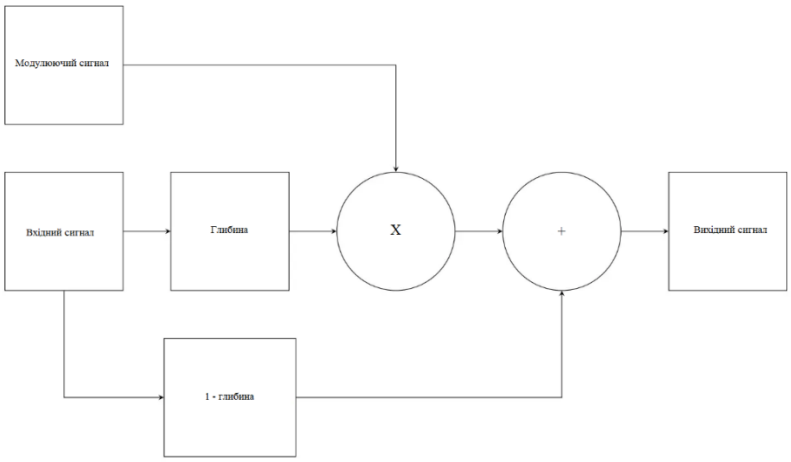

Рисунок 3.9 – Блок-схема ефекту tremolo

На вхід подається семпл, який помножується на глибину ефекту та на значення модулюючого сигналу у цей момент часу, а потім результат додається до значення вхідного семпла, помноженого на 1 – глибина, що змінює амплітуду сигналу в часі.

Структура tremolo має такий вигляд:

```
typedef struct{
	float depth;
	float counter;
	float counter_limit;
	float f_oscillator;
	float f_sample;
	int counter_direction;
}Tremolo;
```

А вихідний семпл розраховується таким чином:

```
int tremolo_calculation(Tremolo *tremolo, int input)
{
    float lfo = (float)tremolo->counter / (float)tremolo-
>counter_limit;
    float gain = (1.0f - tremolo->depth) + tremolo->depth * lfo;

    int output = (int)(input * gain);
    tremolo->counter += tremolo->counter_direction;
      if(tremolo->counter >= tremolo->counter_limit)
      {
            tremolo->counter_direction = -1;
      }
      else if(tremolo->counter <= 0)
      {
            tremolo->counter_direction = 1;
      }
      return output;
}
```

3.5. Програмна реалізація ефекту delay

Delay – це ефект, який зберігає значення вхідного сигналу у буфер, та через деякий час (час затримки), відправляє цей сигнал до виходу. Для реалізації цього ефекту використовуються кільцеві буфери. Блок-схема цього ефекту наведена на рис. 3. 10

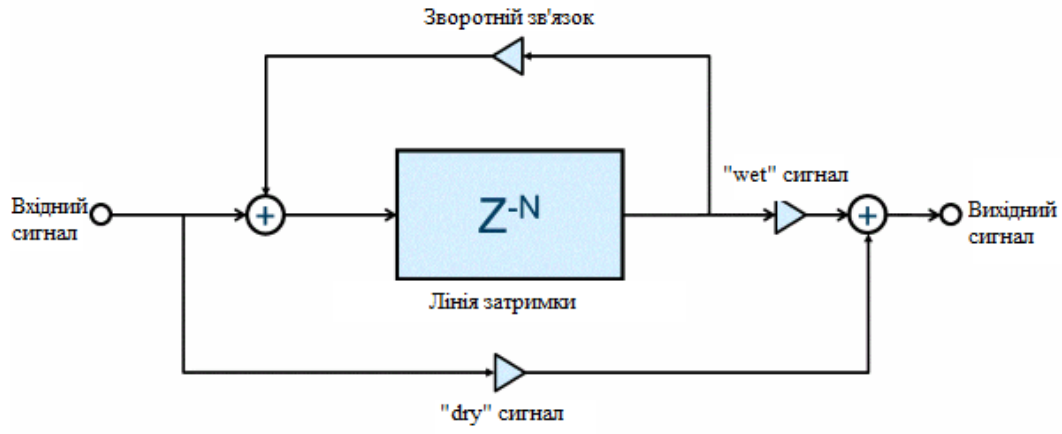

Рисунок 3.10 – Блок-схема ефекту delay

Цей ефект має лінію зворотного зв’язку, яка повертає частину вихідного сигналу до вхідного, що створює ефект затухання. А вихідний сигнал дорівнює сумі “wet”, тобто промодульованого сигналу, та “dry”, тобто сигналу без модуляції, співвідношення яких визначається параметром глибини сигналу.

Таким чином виконується delay у моєму пристрої:

```
float delay(float input)
{
	float output = delay_buffer[delay_index];
	delay_buffer[delay_index] = input + 0.3*output;
	delay_index++;
	if(delay_index == buffer_length)
	{
		delay_index = 0;
	}
	output = (float)input*delay_depth+(float)output*delay_depth;
	return output;
}
```

3.6. Програмна реалізація ефекту chorus

Ефект chorus моделює звучання одночасної гри кількох інструментів, що виконують один і той самий сигнал, але з невеликими часовими та фазовими відхиленнями. Основна ідея полягає в тому, що до основного сигналу додається його ж копія, затримана на змінний у часі інтервал, причому ця зміна задається низькочастотною осциляцією (LFO).

Як і в ефекті delay використовується кільцевий буфер, в який безперервно записуються вхідні семпли.

Отриманий промодульований сигнал змішується з прямим сигналом, а співвідношення цих двох компонентів регулюються параметром глибини сигналу. Блок-схема цього ефекту наведена на рис. 3.11.

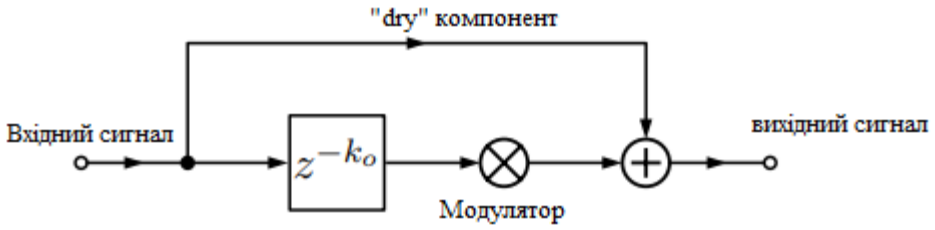

Рисунок 3.11 – Блок-схема ефекта chorus

Структура chorus має такий вигляд:

```
typedef struct{
	int base_delay;
	float changing_delay;
	float total_delay;
	float depth;
	float f_oscillator;
	float f_sample;
	float phase;
	int delay_buf[2000];
}Chorus;
```

Ініціалізація:

```
void chorus_init(Chorus *chorus, float base_delay, float depth, float
f_oscillator, float f_sample)
{
	chorus->base_delay = base_delay;
	chorus->depth = depth;
	chorus->f_oscillator = f_oscillator;
	chorus->f_sample = f_sample;
	chorus->changing_delay = 0;
	chorus->phase = 0;
	for(int i = 0; i<2000; i++)
	{
		chorus->delay_buf[i] = 0;
	}
}
```

Розрахунок вихідного семплу:

```
float chorus_processing(Chorus *chorus, float input)
{
	chorus->delay_buf[write_position] = input;
	chorus->phase += 2*M_PI*(chorus->f_oscillator/chorus->f_sample);
	if(chorus->phase >= 2*M_PI)
	{
		chorus->phase -= 2*M_PI;
	}
	float base_delay = chorus->base_delay*chorus->f_sample/1000.0f;
	float depth = chorus->depth*chorus->f_sample/1000.0f;
	chorus->changing_delay = sinf(chorus->phase)*depth;
	chorus->total_delay = chorus->changing_delay+base_delay;
```

```
        float read_position = (float)write_position-chorus->total_delay;
        if(read_position<0)
        {
              read_position += 2000;
        }
        int i = (int)read_position;
        int j = i+1; if(j == 2000) j = 0;
        float frac = read_position-i;
        float output = 0.75f*input+0.25f*(frac*chorus->delay_buf[j]+(1-
frac)*chorus->delay_buf[i]);
        write_position++;
        if(write_position >= 2000) write_position = 0;
        return output;
}
```

Виходить так, що індекси, з яких виконується зчитування, не є цілими числами. Виходить індекс такого вигляду:

$$i.frac = i + frac$$

Тому, для запобігання спотворень і зчитування проміжних значень виконується інтерполяція. І через свою простоту у реалізації та швидкодію була обрана лінійна інтерполяція.

$$y[n] = frac * x[i+1] + (1 - frac) * x[i]$$

Це робить роботу алгоритму більш плавною.

3.7. Програмна реалізація ефекту overdrive

Ефект overdrive полягає у тому, що вхідний сигнал обрізається (виконується кліппінг), якщо його значення перевищує певний поріг. Є два види кліппінгу – жорсткий та м’який.

Жорсткий кліппінг змушує сигнал, що перевищує порогове значення, приймати це значення, що приводить до більш різких зрізів, і це генерує більше гармонік, що може погано вплинути на звучання, але також

М’який кліппінг відбувається тоді, коли вхідний сигнал стає більшим за визначене порогове значення, і та частина сигналу, що виявляється вище порогу, стискається у верхній частині таким чином, що піки, які були над порогом, зменшуються і повертаються нижче цього порогу.

У цій роботі використовується такий алгоритм м’якого кліппінгу:

$$f(x) = \begin{cases} 2x & for\ x \le \frac{1}{3} \\ -3x^2 + 4x - \frac{1}{3} & for\ \frac{1}{3} < x \le \frac{2}{3} \\ 1 & for\ x > \frac{2}{3} \end{cases}$$

При значеннях менших за 0, у цій функції змінюється знак. Графік цієї функції зображений на рис. 3.12.

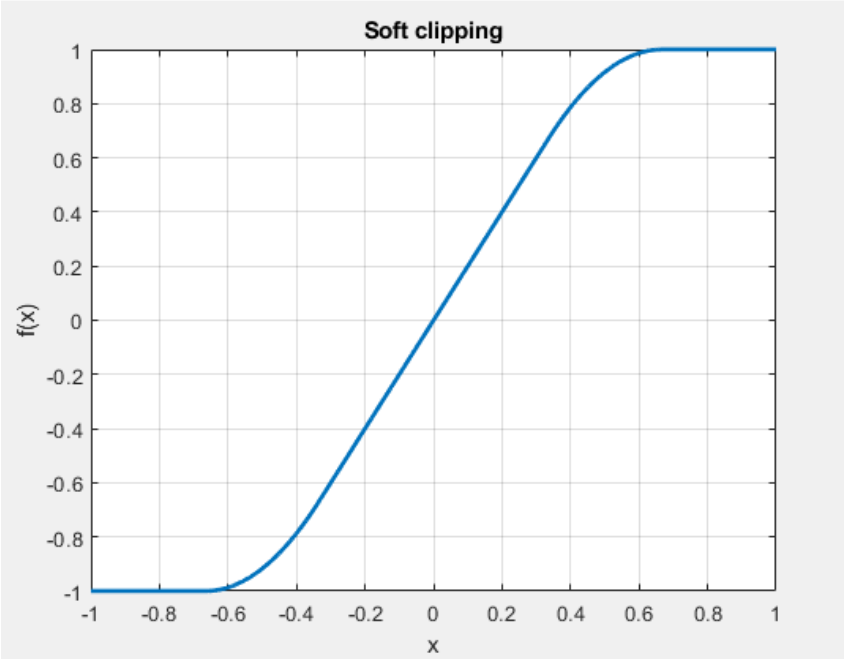

Рисунок 3.12 – М’який кліппінг

Перед тим, як виконувати overdrive, потрібно відфільтрувати низькочастотний компонент сигналу, який буде погано звучати при перегрузі.

```
HPF_input[1] = input-3050;
HPF_output[1] = HPF_coefficient*(HPF_input[1]-HPF_input[0]+HPF_output[0]);
HPF_output[0] = HPF_output[1];
HPF_input[0] = HPF_input[1];
signal  = HPF_output[1];
```

Де

```
HPF_coefficient = 1.0f/(1.0f+alpha);
```

```
alpha = 2.0f*M_PI*HPF_f0/SAMPLING_FREQUENCY;
```

Програмно overdrive реалізован таким чином:

```
int overdrive(int input, int threshold, int gain)
{
```

```
        int sign = 1;
        float f;
        int output;
        if(input < ADC_medium)
        {
              sign = -1;
        }
        int signal = abs(input);
        float ratio = gain*(float)signal/(float)threshold;
        if(ratio <= 1.0f/3.0f)
        {
              f = 2.0f*ratio;
        }
        else if(ratio > 1.0f/3.0f && ratio <= 2.0f/3.0f)
        {
              f = -3.0f*ratio*ratio+4.0f*ratio-1.0f/3.0f;
        }
        else
        {
              f = 1.0f;
        }
        output = f*sign*(float)threshold;
        return output;

}
```

Ефект overdrive створює гармоніки високої частоти, тому існує потреба відфільтрувати їх.

Для цього використовується FIR (Finite Impulse Response) фільтр, тобто фільтр зі скінченною імпульсною характеристикою.

Ці фільтри мають такий вигляд:

$$y[n] = \sum_{i=0}^{N} b_i * x[n-i]$$

Їхня особливість в тому, що вони на відміну від IIR фільтрів, залежать лише від вхідних семплів. Але також вони потребують більших потужностей для розрахунків. Тому треба знайти компроміс між бажаною АЧХ та кількістю «тапів» у цьому фільтрі.

АЧХ та імпульсна характеристика застосованого мною фільтру зображені на рис. 3.13 та 3.14.

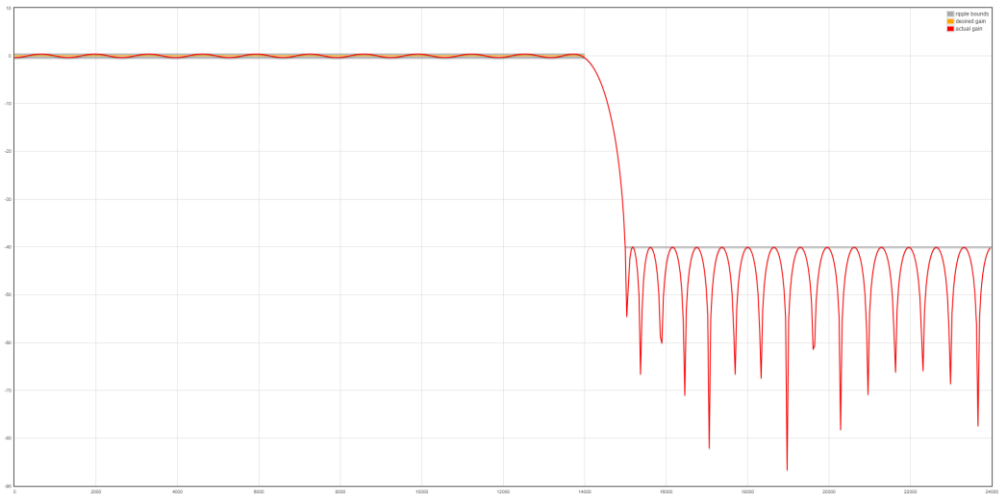

Рисунок 3.13 – АЧХ фільтру

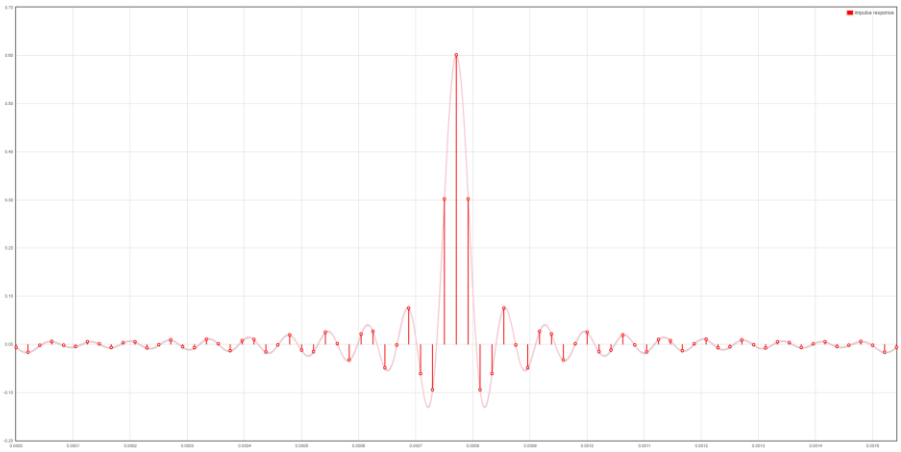

Рисунок 3.14 – Імпульсна характеристика фільтру

Цей фільтр також реалізується завдяки кільцевому буферу:

```
float filtration(float input)
{
      filter_buf[filter_buf_index] = input;
    filter_buf_index++;
    if(filter_buf_index == FILTER_LENGTH)
    {
      filter_buf_index = 0;
    }
    filter_output = 0.0f;
    int sum_index = filter_buf_index;
    for(int j = 0; j<FILTER_LENGTH; j++)
    {
       if(sum_index > 0)
       {
         sum_index--;
       }
```

```
            else
            {
              sum_index = FILTER_LENGTH-1;
            }
            filter_output += impulse_response[j] * filter_buf[sum_index];
        }
        return filter_output;
    }
```

3.8. Керування пристроєм

Для того, щоб виводити інформацію про налаштування ефектів, використовується дисплей SSD1306. На ньому відображається панель меню, яка оновлюється тільки тоді, коли виконуються якісь зміни.

```
if(audio_flag == 1 && menu_redraw_flag == 1)
        {
              if(menu_screen == 1)
              {
                    if(menu_page == 0)
                    {
                          sprintf(text_buffer, "delay %s", delay_flag ? "on"
: "off");
                          draw_menu_line(text_buffer, 1);
                          sprintf(text_buffer, "overdrive %s",
overdrive_flag ? "on" : "off");
                          draw_menu_line(text_buffer, 2);
                          sprintf(text_buffer, "tremolo %s", tremolo_flag ?
"on" : "off");
                          draw_menu_line(text_buffer, 3);
                          sprintf(text_buffer, "chorus %s", chorus_flag ?
"on" : "off");
                          draw_menu_line(text_buffer, 4);

                    }
```

І так далі.

Перехід між сторінками меню виконується за допомогою кнопок. Дві кнопки відповідають за перехід вниз – уверх по строчкам, а одна відповідає за вмикання/вимикання ефектів по одному натисканню кнопки, або перехід у налаштування по довгому натисканню.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_PIN)
{
      static uint32_t press = 0;
      uint32_t now = HAL_GetTick();
      if(GPIO_PIN == GPIO_PIN_10)
      {
            uint8_t state = HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_10);
          if (state == GPIO_PIN_SET)
          {
              btn_is_pressed = 1;
              btn_press_time = now;
```

```
            }
            else
            {
                uint32_t duration = HAL_GetTick() - btn_press_time;

                if (duration < 200)
                {
                     if(menu_page == 0 && menu_line == 1) delay_flag = !delay_flag;
                     if(menu_page == 0 && menu_line == 2) overdrive_flag =
!overdrive_flag;
                     if(menu_page == 0 && menu_line == 3) tremolo_flag =
!tremolo_flag;
                     if(menu_page == 0 && menu_line == 4) chorus_flag = !chorus_flag;

                }
                else
                {
                     menu_flag = !menu_flag;
                    menu_page = menu_flag*menu_line;
                }

                btn_is_pressed = 0;
            }
        }
        if((now - press) > 200)
        {

              if(GPIO_PIN == GPIO_PIN_12)
              {
                    menu_line--;
                    if(menu_line < 1)
                    {
                          if(menu_screen == 2)
                          {
                                menu_line = 4;
                                menu_screen = 1;
                          }
                          else
                          {
                                menu_line = 1;
                          }
                    }
              }

              else if(GPIO_PIN == GPIO_PIN_11)
              {
                    menu_line++;
                    if(menu_line > 4 && menu_screen == 1)
                    {
                          menu_screen = 2;
                          menu_line = 1;
                    }
                    else if(menu_line > 3 && menu_screen == 2)
                    {
                          menu_line = 3;
                    }
              }
        }
        menu_redraw_flag = 1;
        press = now;
}
```

Потенціометром виконується зміна налаштувань, яка виконується тільки тоді, коли значення нового зчитування АЦП більша за попередню, щоб уникнути раптових змін.

```
	else if(hadc == &hadc2 && abs(old_adc - menu_control[0]) > 100 && audio_flag
== 1)
	{
		if(menu_screen == 1)
		{
			if(menu_page == 1)
			{
				if(menu_line == 1)
				{
					delay_time = menu_control[0]/16;
				}
				else
				{
					delay_depth = (float)menu_control[0]/4096.0f;
				}
			}
```

3.9. Результати

Для перевірки результатів на вхід пристрою подається синусоїда з амплітудою 0.5 В.

Зовнішній вигляд меню зображений на рис 3.15 та 3.16.

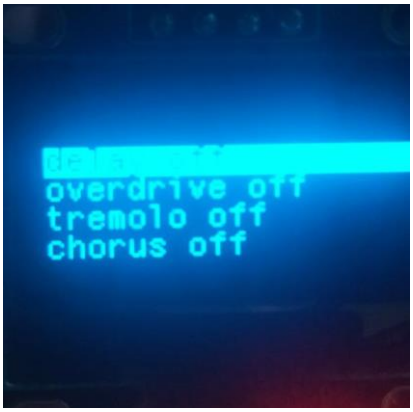

Рисунок 3.15 – Перша сторінка меню

Рисунок 3.16 – Друга сторінка меню

Вхідний сигнал зображений на рис. 3.17.

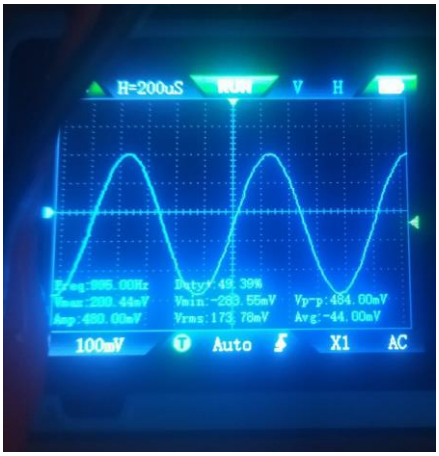

Рисунок 3.17 – Сигнал без обробки

Вихідний сигнал після застосування на ньому ефекту overdrive наведений на рис 3.18.

Рисунок 3.18 – Сигнал після ефекту overdrive

Вихідний сигнал після застосування на ньому ефекту tremolo наведений на рис 3.19. Був збільшений час вибірки, щоб краще побачити вплив ефекту.

Рисунок 3.19 – Сигнал після ефекту tremolo

Вихідний сигнал після застосування на ньому ефекту chorus наведений на рис 3.19.

Рисунок 3.20 – Сигнал після ефекту chorus

Вхідний сигнал та вихідний сигнал після проходження low-shelf фільтра зображені на рис. 3.21 та рис. 3.22.

Рисунок 3.21 – Сигнал низької частоти до проходження low-shelf фільтра

Рисунок 3.22 – Сигнал низької частоти після проходження low-shelf фільтра

Вхідний сигнал та вихідний сигнал після проходження peak фільтра зображені на рис. 3.23 та рис. 3.24.

Рисунок 3.23 – Сигнал середньої частоти до проходження peak фільтра

Рисунок 3.24 – Сигнал середньої частоти після проходження peak фільтра

Вхідний сигнал та вихідний сигнал після проходження high-shelf фільтра зображені на рис. 3.23 та рис. 3.24.

Рисунок 3.25 – Сигнал високої частоти до проходження high-shelf фільтра

Рисунок 3.26 – Сигнал високої частоти після проходження high-shelf фільтра

3.10. Висновки до третього розділу

У цій главі були розглядані принципи реалізації деяких гітарних ефектів та фільтрів методами цифрової обробки сигналів. Було організовано безперервну обробку семплів завдяки DMA та двійної буферизації, було реалізовано структуру біквадратного фільтру, а також реалізовано low-shelf, peak, та high-shelf фільтри, що застосовуються у всіх еквалайзерах.

Було реалізовано ефекти delay, chorus. Була застосована лінійна інтерполяція для отримання менш спотвореного сигналу. Було розроблено реалізовано ефект tremolo та м’який кліпінг для ефекта overdrive. Також було розроблено просте меню та панель керування параметрами цих ефектів. Була отримана працездатна система.

					171.6321М.КР.ПЗ.Р4	
Змн.	*Арк.*	*№ документу*	*Підпис*	*Дата*		
Розробник		Мосійчук К.С.			Охорона праці	*Літ.* / *Арк.* 54 / *Аркушів*
Консультант		Тубальцев. А.М.				
Н. контр.		Добрінова Л.П.				
Керівник		Дьяконов О.С.				НУК імені адмірала Макарова
Зав. каф.		Рябенький В. М.				

РОЗДІЛ 4. ОХОРОНА ПРАЦІ

4.1. Вступ

Охорона праці займає важливе місце у забезпеченні безпеки та здоров'я працівників. Будь-яка діяльність, пов'язана з розробкою, виробництвом та використанням електронних пристроїв, вимагає суворого дотримання правил і норм охорони праці.

Державна політика в галузі охорони праці визначається відповідно до Конституції України Верховною Радою України і спрямована на створення належних, безпечних і здорових умов праці, запобігання нещасним випадкам та професійним захворюванням. Одним з принципів є підвищення рівня промислової безпеки шляхом забезпечення суцільного технічного контролю за станом виробництв, технологій та продукції, а також сприяння підприємствам у створенні безпечних та нешкідливих умов праці.

У цьому розділі розглянуті захист від радіовипромінювання, заходи для зменшення впливу радіовипромінювання на здоров'я працівників, та заходи для запобігання ураження електричним струмом, вимоги до ізоляції та захисту електричних з'єднань і компонентів.

4.2. Небезпека, викликана ураженням електричним струмом

Електротравми виникають при дії електричного струму на організм. Вони мають багато причин, найчастіше ці причини суто організаційні. Ризик ураження електричним струмом збільшується при таких умовах:

- Відносна вологість повітря більше 75 відсотків
- Велика кількість технологічного, металевого пилу у повітрі
- У приміщені виробництва струмопровідна підлога

Електричний струм може діяти на організм людини багатьма способами. Це – термічна дія, що призводить до виникнення опіків, електролітична

дія, що розкладає органічні речовини у тілі людини, а також крові, і призводить до порушення їхнього фізико-хімічного складу, металізація

шкіри, тобто проникнення у верхні шари шкіри металевих частинок, що розплавилися від дії електричної дуги, біологічна дія, що призводить до скорочень м’язів і порушень у диханні та роботі серця.

Є два типи чинників, що впливають на те, наскільки сильним буде ураження електричним струмом: електричного та неелектричного характеру. До чинників електричного характеру належать сила струму та напруга, частота, електричний опір людського тіла.

Неелектричні чинники ж це тривалість дії струму, шлях його проходження крізь тіло людини, умови навколишнього середовища. Також, психофізіологічний стан людини також має значення. Встановлено, що людина, яка підготовлена до сприйняття електричного удару, підвергається меншим степенем небезпеки, ніж людина, для якої цей удар був несподіваним.

Змінний струм є більш небезпечним ніж постійний, це видно зі значень уражаючих струмів.

За ступенем фізіологічного впливу можна виділити такі діапазони уражаючих струмів (значення змінного струму приводяться при частоті 50 Гц):

- 0.6-1.5 мА при змінному і 5-7 мА при постійному струмі – пороговий відчутний струм, це значення людина вже починає відчувати
- 6-10 мА змінного струму та 30-40 мА постійного – відпускаючий струм, при якому все ще можливо відірвати від струмопровідних частин, хоча і з великими зусиллями. При цих значеннях струму з’являються спазми у м’язах та біль у руці.
- 10-15 мА змінного струму та 50-80 мА постійного – пороговий приковуючий струм, при якому вже для людини неможливо самостійно звільнитися від струмопровідних частин.
- 100 мА змінного струму та 300 мА постійного – пороговий фібріляційний струм. Дія цього струму викликає зупинку дихання та

хаотичне скорочення волокон серцевого м’язу, що не дає серцю переміщувати кров по судинах. При струму більше 5 А, фібріляції не виникає, серце зупиняється одразу.

4.3. Захист від ураження електричним струмом

Основними заходами захисту від ураження електричним струмом є:

- Ізоляція струмопровідних частин, для захисту від випадкового дотику до них
- Використання малих значень напруги
- Захисне заземлення та занулення
- Застосування автоматичних вимикачів та запобіжників
- Застосування індивідуальних засобів захисту

Ізоляція провідників забезпечується покриттям їх поверхонь діелектриком. Ізоляція буває трьох типів: робоча, додаткова та подвійна. Робоча ізоляція забезпечує захист людини від ураження струмом, додаткова накладається з метою захисту людини при пошкоджені робочої ізоляції, а подвійна складається з цих двох типів.

Малі значення напруги використовуються для того, щоб зробити струм, що проходить крізь тіло людини, відносно безпечним. Однак, використання малих значень напруги не гарантує повну безпеку, і тому цей метод не використовується поодинці. Такий метод використовується здебільше у переносних пристроях.

Захисне заземлення – з’єднання зі землею металевих частин електроустановки, що не перебувають під напругою, але можуть опинитися під нею під час аварії. З таким з’єднанням, людина буде захищена від ураження струмом при доторканні до таких частин. Занулення – це з’єднання таких частин з нульовим захисним проводом.

Також має сенс використання антистатичних браслетів, килимків та інших засобів для захисту від статичної електрики, що може пошкодити електронні компоненти пристрою, а також забезпечення належної вологості повітря в приміщеннях для зменшення накопичення статичної електрики.

4.4. Електромагнітне поле

Електромагнітне поле, що створюється зчитувачем та радіомодулем, також може впливати на організм людини.

Тривалий вплив високих рівнів ЕМП може мати наступні негативні наслідки:

- Головний біль та втома
- Погіршення концентрації
- Порушення сну
- Вплив на серцево-судинну систему
- Можливий ризик розвитку онкологічних захворювань при довгому впливі.

Існує також тепловий вплив ЕМП на тіло людини, що проявляється у нагріві певних тканин тіла при переході електромагнітної енергії у теплову.

Захист від електромагнітного випромінювання може здійснюватися по-різному. Перебування персоналу у зоні дії ЕМП може бути обмеженим невеликим проміжком часу, джерело ЕМП може бути розташовано на далекій відстані від персоналу. Також, джерела випромінювання можуть бути екрановані, що є найбільш ефективним методом захисту. Для виготовлення екрану використовуються матеріали з високою електропровідністю: мідь, латунь, алюміній, сталь. З цих матеріалів формують металевий лист або сітку та під'єднують до заземлення. Енергія випромінювання поглинається цим екраном і відводиться у землю, або відбивається від екрану.

ВИСНОВКИ

У цій роботі був проведений повний цикл розробки гітарного процесору. Була розроблена аналогова частина, яка забезпечує низький шум системи завдяки вхідним та вихідним фільтрам та буферам, а також розділенню живлення цифрової та аналоговий частини з застосуванням малошумного лінійного перетворювача напруги для аналогової частини.

Подвійна буферизація при використанні DMA у АЦП та ЦАП мікроконтролера дають можливість оброблювати сигнал у реальному часі.

Були імплементовані та налагоджені такі ефекти як delay, chorus, tremolo, overdrive. Був розроблений трьохсмуговий еквалайзер, що підсилює сигнал у трьох частотних діапазонах: низькому, середньому, високому.

Була розроблена панель керування та дисплей з меню, які дають можливість користувачу змінювати легко та гнучко змінювати параметри ефектів.

Однак, треба зазначити, що цей пристрій можна покращити. Наприклад, застосувати для зчитування аналогового сигналу не вбудований АЦП мікроконтролера, а мікросхему аудіокодека, яку можно підключити до мікроконтролера за допомогою інтерфейсів I2C для керування ним, та I2S для прийому самого аудіосигналу. Такі мікросхеми дають більшу розрядність АЦП, що значить і більшу точність, та більшу частоту семплінгу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Contributors to Wikimedia projects. Electric guitar - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Electric_guitar (дата доступу: 07.12.2025).
2. Contributors to Wikimedia projects. Distortion (music) - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Distortion_(music) (дата доступу: 07.12.2025).
3. Audio EQ Cookbook. W3C. URL: https://www.w3.org/TR/audio-eq-cookbook/ (date of access: 07.12.2025).
4. Dattoro J. Effect design part 2: delay-line modulation and chorus. J. audio eng sot. 1997. Vol. 45, no. 10.
5. Contributors to Wikimedia projects. Finite impulse response - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Finite_impulse_response (дата доступу: 09.12.2025).
6. Contributors to Wikimedia projects. Digital biquad filter - Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Digital_biquad_filter (дата доступу: 09.12.2025).
7. Boquera G. A. Implement a chorus effect in a real-time embedded system. Барселона, 2016 р. 60 стр.
8. Åberg B. Low-Latency Digital Guitar Effects Using Signal Processing with Python in Real Time. Турку, 2024 р. 76 стр.
9. Self D. Small Signal Audio Design. Taylor & Francis Group, 2014.
10. Zeki E. DIGITAL MODELLING OF GUITAR AUDIO EFFECTS : Master thesis. Анкара, 2015 р. 148 стр.
11. Електробезпека на підприємстві / Електробезпека на виробництві / Електробезпека охорона праці. Довідник спеціаліста з охорони праці.

URL: https://pro-op.com.ua/article/745-elektrobezpeka (дата звернення: 09.12.2025).

12. Електробезпека. Відокремлений структурний підрозділ "Житомирський торговельно-економічний фаховий коледж Державного торговельно-економічного університету". URL: http://www.ztec.com.ua/ztec/e-lib/Охорона%20праці/Тема%2017%20Електробезпека.pdf (дата звернення: 09.12.2025).
13. Захист від електромагнітних випромінювань радіочастот. Довідник спеціаліста з охорони праці. URL: https://pro-op.com.ua/article/725-zahist-vd-elektromagntnogo-vipromnyuvannya-radochastot (дата звернення: 09.12.2025).
14. Відокремлений структурний підрозділ "Житомирський торговельно-економічний фаховий коледж Державного торговельно-економічного університету". URL: http://www.ztec.com.ua/ztec/e-lib/Охорона%20праці/Тема%2012%20ЕМП,%20випромінювання%20оптич%20діапазону.pdf (дата звернення: 09.12.2025)

ДОДАТОК

Текст програми для гітарного процесора

```
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "math.h"
#include "stdio.h"
#include "stdlib.h"
#include "fonts.h"
#include "ssd1306.h"
#include "equalizer.h"
#include "tremolo.h"
#include "chorus.h"
#define FILTER_LENGTH 61
#define BUFFER_SIZE 128
#define SAMPLING_FREQUENCY 45000
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc1;
ADC_HandleTypeDef hadc2;
DMA_HandleTypeDef hdma_adc1;
DMA_HandleTypeDef hdma_adc2;

DAC_HandleTypeDef hdac;
DMA_HandleTypeDef hdma_dac1;

I2C_HandleTypeDef hi2c1;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC1_Init(void);
```

```
static void MX_DAC_Init(void);
static void MX_TIM2_Init(void);
static void MX_ADC2_Init(void);
static void MX_TIM3_Init(void);
static void MX_I2C1_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
Biquad lowshelf;
Biquad highshelf;
Biquad peakingEQ;
Tremolo tremolo;
Chorus chorus;
float filter_buf[FILTER_LENGTH];
float filter_output;
int filter_buf_index;
static float impulse_response[FILTER_LENGTH] = {
              -0.005567617253669317,
               0.006486025115508817,
               0.020758365954009053,
               0.022306457858621564,
               0.005434545965274444,
               -0.009309843519192514,
               -0.00355923773354174,
               0.009623495913482358,
               0.005944201160456905,
               -0.009222982057483599,
               -0.008728461208249928,
               0.008463270228412069,
               0.012124003359155852,
               -0.006813975850490977,
               -0.015892571183819783,
               0.003979736297278519,
               0.019894226551181067,
               0.0004200328958750836,
               -0.023892042489803242,
               -0.006814042655329699,
               0.027685466983796505,
               0.015955760667264087,
               -0.031129433684257635,
               -0.02951581333463094,
               0.0340199696338675,
               0.05180498468135684,
               -0.03617590214091364,
               -0.09880268966958553,
               0.037507367109538996,
               0.31584262882983494,
               0.4620402427907755,
               0.31584262882983494,
               0.037507367109538996,
               -0.09880268966958553,
               -0.03617590214091364,
               0.05180498468135684,
               0.0340199696338675,
               -0.02951581333463094,
               -0.031129433684257635,
               0.015955760667264087,
               0.027685466983796505,
```

```
                -0.006814042655329699,
                -0.023892042489803242,
                0.0004200328958750836,
                0.019894226551181067,
                0.003979736297278519,
                -0.015892571183819783,
                -0.006813975850490977,
                0.012124003359155852,
                0.008463270228412069,
                -0.008728461208249928,
                -0.009222982057483599,
                0.0059442011604568905,
                0.009623495913482358,
                -0.00355923773354174,
                -0.009309843519192514,
                0.005434545965274444,
                0.022306457858621564,
                0.020758365954009053,
                0.006486025115508817,
                -0.005567617253669317
            };
/**
  * @brief  The application entry point.
  * @retval int
  */
uint16_t  adc_buf[BUFFER_SIZE];
uint16_t  dac_buf[BUFFER_SIZE];
uint16_t processor_output;
volatile uint8_t half_flag = 0;
volatile uint8_t full_flag = 0;
volatile float HPF_output[2];
volatile float HPF_input[2];

volatile int overdrive_threshold = 800;
volatile int overdrive_gain = 1;
volatile uint8_t overdrive_flag = 0;

volatile int delay_time = 0;
volatile float delay_depth = 0.0f;
volatile uint8_t delay_flag = 0;

volatile int tremolo_frequency = 0;
volatile float tremolo_depth = 0.0f;
volatile uint8_t tremolo_flag = 0;

volatile  float chorus_base_delay = 20.0f;
volatile float chorus_depth = 5.0f;
volatile float chorus_lfo_freq = 3.0f;
volatile uint8_t chorus_flag = 0;

volatile int lowshelf_frequency = 0;
volatile float lowshelf_s = 0;
volatile int lowshelf_gain = 0;

volatile int highshelf_frequency = 0;
volatile float highshelf_s = 0;
volatile int highshelf_gain = 0;

volatile int peak_frequency = 0;
volatile float peak_q = 0;
volatile int peak_gain = 0;
```

```
float alpha = 2.0f*M_PI*80/SAMPLING_FREQUENCY;
float HPF_coefficient = 0;

uint16_t menu_control[1];
volatile uint8_t menu_line = 1;
volatile uint8_t menu_page = 0;
volatile uint8_t menu_screen = 1;
volatile uint8_t menu_flag = 0;
volatile uint8_t setup_flag = 0;
volatile uint8_t menu_redraw_flag = 1;

volatile uint8_t adc2_flag = 0;
volatile int old_adc = 0;
int audio_flag = 0;

volatile uint32_t btn_press_time = 0;
volatile uint8_t btn_is_pressed = 0;

int delay_buffer[20000];
int delay_index = 0;
int buffer_length = 0;

void delay_init(int ms_length)
{
	buffer_length = 0.001*ms_length/(float)(1/SAMPLING_FREQUENCY);
	for(int i = 0; i<buffer_length; i++)
	{
		delay_buffer[i] = 0;
	}
}
float delay(float input)
{
	float output = delay_buffer[delay_index];
	delay_buffer[delay_index] = input + 0.3*output;
	delay_index++;
	if(delay_index == buffer_length)
	{
		delay_index = 0;
	}
	output = (float)input*delay_depth+(float)output*delay_depth;
	return output;
}
void filter_init()
{
	for(int i = 0; i<FILTER_LENGTH; i++)
	{
		filter_buf[i] = 0.0f;
	}
	filter_buf_index = 0;
	filter_output = 0.0f;
}
float filtration(float input)
{
	filter_buf[filter_buf_index] = input;
    filter_buf_index++;
    if(filter_buf_index == FILTER_LENGTH)
    {
      filter_buf_index = 0;
    }
    filter_output = 0.0f;
```

```
    int sum_index = filter_buf_index;
    for(int j = 0; j<FILTER_LENGTH; j++)
    {
       if(sum_index > 0)
       {
         sum_index--;
       }
       else
       {
         sum_index = FILTER_LENGTH-1;
       }
       filter_output += impulse_response[j] * filter_buf[sum_index];
    }
    return filter_output;
}


int overdrive(int input, int threshold, int gain)
{
      int sign = 1;
      float f;
      int output;
      if(input < 0)
      {
            sign = -1;
      }
      int signal = abs(input);
      float ratio = gain*(float)signal/(float)threshold;
      if(ratio <= 1.0f/3.0f)
      {
            f = 2.0f*ratio;
      }
      else if(ratio > 1.0f/3.0f && ratio <= 2.0f/3.0f)
      {
            f = -3.0f*ratio*ratio+4.0f*ratio-1.0f/3.0f;
      }
      else
      {
            f = 1.0f;
      }
      output = f*sign*(float)threshold;
      output += ADC_medium;
      return output;

}

int process_half_buffer(int input)
{
      int signal = 0;
      HPF_input[1] = input-3050;
      HPF_output[1] = HPF_coefficient*(HPF_input[1]-HPF_input[0]+HPF_output[0]);
      HPF_output[0] = HPF_output[1];
      HPF_input[0] = HPF_input[1];
      signal  = HPF_output[1];
      signal = biquad_calculation(&lowshelf, HPF_output[1]);
      signal = biquad_calculation(&highshelf,  HPF_output[1]);
      signal = biquad_calculation(&peakingEQ, HPF_output[1]);
      if(overdrive_flag == 1)
      {
            signal = filtration(overdrive(signal, overdrive_threshold,
overdrive_gain));
```

```
	}
	if(delay_flag == 1)
	{
		if(adc2_flag) delay_init(delay_time);
		signal = delay(processor_output);
	}
	if(tremolo_flag == 1)
	{
		if(adc2_flag) tremolo_init(&tremolo, tremolo_frequency,
SAMPLING_FREQUENCY, tremolo_depth);
		signal = tremolo_calculation(&tremolo, signal);
	}
	if(chorus_flag == 1)
	{
		if(adc2_flag) chorus_init(&chorus, chorus_base_delay, chorus_depth,
chorus_lfo_freq, SAMPLING_FREQUENCY);
		signal = chorus_processing(&chorus, signal);
	}

	processor_output = signal;
	return processor_output+3050;

}
void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef *hadc)
{
	if(hadc == &hadc1)
	{
		half_flag = 1;
	}
}
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
	if(hadc == &hadc1)
	{
		full_flag = 1;

	}
	else if(hadc == &hadc2 && abs(old_adc - menu_control[0]) > 100 && audio_flag
== 1)
	{
		if(menu_screen == 1)
		{
			if(menu_page == 1)
			{
				if(menu_line == 1)
				{
					delay_time = menu_control[0]/16;
				}
				else
				{
					delay_depth = (float)menu_control[0]/4096.0f;
				}
			}
			else if(menu_page == 2)
			{
				if(menu_line == 1)
				{
					overdrive_threshold = menu_control[0]/5;
				}
				else
				{
```

```
                        overdrive_gain = menu_control[0]/1000+1;
                }
        }
        else if(menu_page == 3)
        {
                if(menu_line == 1)
                {
                        tremolo_frequency = menu_control[0]/275;
                }
                else
                {
                        tremolo_depth = (float)menu_control[0]/4096.0f;
                }
        }
        else if(menu_page == 4)
        {
                if(menu_line == 1)
                {
                        chorus_base_delay = menu_control[0]/136;
                }
                else if(menu_line == 2)
                {
                        chorus_depth = (float)menu_control[0]/409.6f;
                }
                else
                {
                        chorus_lfo_freq = (float)menu_control[0]/2040.0f;
                }
        }
}
else if(menu_screen == 2)
{
        if(menu_page == 1)
        {
                if(menu_line == 1)
                {
                        lowshelf_frequency = 10+menu_control[0]/30;
                }
                else if(menu_line == 2)
                {
                        lowshelf_s = 0.7+(float)menu_control[0]/8090.0f;
                }
                else
                {
                        lowshelf_gain = menu_control[0]/409.6;
                }
        }
        else if(menu_page == 2)
        {
                if(menu_line == 1)
                {
                        peak_frequency = 500+menu_control[0]/10;
                }
                else if(menu_line == 2)
                {
                        peak_q = 0.5+(float)menu_control[0]/1000;
                }
                else
                {
                        peak_gain = menu_control[0]/409.6;
                }
```

```
                }
                else if(menu_page == 3)
                {
                        if(menu_line == 1)
                        {
                                highshelf_frequency = 5000+menu_control[0];
                        }
                        else if(menu_line == 2)
                        {
                                highshelf_s = 0.7+(float)menu_control[0]/8090.0f;
                        }
                        else
                        {
                                highshelf_gain = menu_control[0]/409.6;
                        }
                }
            }
            adc2_flag = 1;
            menu_redraw_flag = 1;
            old_adc = menu_control[0];
        }
}


void HAL_GPIO_EXTI_Callback(uint16_t GPIO_PIN)
{
        static uint32_t press = 0;
        uint32_t now = HAL_GetTick();
        if(GPIO_PIN == GPIO_PIN_10)
        {
              uint8_t state = HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_10);
            if (state == GPIO_PIN_SET)
            {
                btn_is_pressed = 1;
                btn_press_time = now;
            }
            else
            {
                uint32_t duration = HAL_GetTick() - btn_press_time;

                if (duration < 200)
                {
                     if(menu_page == 0 && menu_line == 1) delay_flag = !delay_flag;
                     if(menu_page == 0 && menu_line == 2) overdrive_flag =
!overdrive_flag;
                     if(menu_page == 0 && menu_line == 3) tremolo_flag =
!tremolo_flag;
                     if(menu_page == 0 && menu_line == 4) chorus_flag = !chorus_flag;

                }
                else
                {
                     menu_flag = !menu_flag;
                    menu_page = menu_flag*menu_line;
                }

                btn_is_pressed = 0;
            }
        }
        if((now - press) > 200)
        {
```

```
			if(GPIO_PIN == GPIO_PIN_12)
			{
				menu_line--;
				if(menu_line < 1)
				{
					if(menu_screen == 2)
					{
						menu_line = 4;
						menu_screen = 1;
					}
					else
					{
						menu_line = 1;
					}
				}
			}

			else if(GPIO_PIN == GPIO_PIN_11)
			{
				menu_line++;
				if(menu_line > 4 && menu_screen == 1)
				{
					menu_screen = 2;
					menu_line = 1;
				}
				else if(menu_line > 3 && menu_screen == 2)
				{
					menu_line = 3;
				}
			}
		}
		menu_redraw_flag = 1;
		press = now;
}

void draw_menu_line(char *string, int line)
{
		int y = 7+10*line;
		if(menu_line == line)
		{
			SSD1306_DrawFilledRectangle(0, y, 128, 10, SSD1306_COLOR_WHITE);
			SSD1306_GotoXY(0, y);
			SSD1306_Puts(string, &Font_7x10, SSD1306_COLOR_BLACK);
		}
		else
		{
			SSD1306_DrawFilledRectangle(0, y, 128, 10, SSD1306_COLOR_BLACK);
			SSD1306_GotoXY(0, y);
			SSD1306_Puts(string, &Font_7x10, SSD1306_COLOR_WHITE);
		}
}

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
```

```
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC1_Init();
  MX_DAC_Init();
  MX_TIM2_Init();
  MX_ADC2_Init();
  MX_TIM3_Init();
  MX_I2C1_Init();
  /* USER CODE BEGIN 2 */
  char text_buffer[25];
  SSD1306_Init();
  delay_init(500);
  HPF_coefficient = 1.0f/(1.0f+alpha);
  HPF_output[0] = 0; HPF_output[1] = 0;
  HPF_input[0] = 0; HPF_input[1] = 0;
  filter_init();
  HAL_TIM_Base_Start(&htim2);
  HAL_TIM_Base_Start(&htim3);
  lowshelf_calculation(&lowshelf, 45000, 200, 1, 0);
  highshelf_calculation(&highshelf, 45000, 4000, 1, 0);
  peak_calculation(&peakingEQ, 45000, 750, 0.7, 0);
  tremolo_init(&tremolo, 12, SAMPLING_FREQUENCY, 0.5);
  chorus_init(&chorus, 20, 5, 3, SAMPLING_FREQUENCY);
  for(int i = 0; i < BUFFER_SIZE; i++)
  {
      dac_buf[i] = 2048;
  }
  HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buf, BUFFER_SIZE);
  HAL_ADC_Start_DMA(&hadc2, (uint32_t*)menu_control, 1);
  HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t*)dac_buf, BUFFER_SIZE,
DAC_ALIGN_12B_R);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
        if(half_flag == 1)
        {
```

```
                audio_flag = 0;
                half_flag = 0;
                for(int i = 0; i<BUFFER_SIZE/2; i++)
                {
                    dac_buf[i] = process_half_buffer(adc_buf[i]);
                }
                audio_flag = 1;
        }
        else if(full_flag == 1)
        {
                audio_flag = 0;
                full_flag = 0;
                for(int i = BUFFER_SIZE/2; i<BUFFER_SIZE; i++)
                {
                    dac_buf[i] = process_half_buffer(adc_buf[i]);
                }
                audio_flag = 1;
        }
        if(audio_flag == 1 && menu_redraw_flag == 1)
        {
                if(menu_screen == 1)
                {
                        if(menu_page == 0)
                        {
                                sprintf(text_buffer, "delay %s", delay_flag ? "on" :
"off");
                                draw_menu_line(text_buffer, 1);
                                sprintf(text_buffer, "overdrive %s", overdrive_flag ?
"on" : "off");
                                draw_menu_line(text_buffer, 2);
                                sprintf(text_buffer, "tremolo %s", tremolo_flag ? "on" :
"off");
                                draw_menu_line(text_buffer, 3);
                                sprintf(text_buffer, "chorus %s", chorus_flag ? "on" :
"off");
                                draw_menu_line(text_buffer, 4);

                        }
                        else if(menu_page == 1)
                        {
                                delay_init(delay_time);
                                sprintf(text_buffer, "delay: %d ms", delay_time);
                                draw_menu_line(text_buffer, 1);
                                sprintf(text_buffer, "depth = %d.%d", (int)delay_depth,
(int)(delay_depth*100));
                                draw_menu_line(text_buffer, 2);
                                SSD1306_DrawFilledRectangle(0, 7+10*3, 128, 10,
SSD1306_COLOR_BLACK);
                                SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                        }
                        else if(menu_page == 2)
                        {
                                sprintf(text_buffer, "threshold: %d",
overdrive_threshold);
                                draw_menu_line(text_buffer, 1);
                                sprintf(text_buffer, "gain: %d", (int)overdrive_gain);
                                draw_menu_line(text_buffer, 2);
                                SSD1306_DrawFilledRectangle(0, 7+10*3, 128, 10,
SSD1306_COLOR_BLACK);
```

```
                        SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                    }
                    else if(menu_page == 3)
                    {
                        tremolo_init(&tremolo, tremolo_frequency,
SAMPLING_FREQUENCY, tremolo_depth);
                        sprintf(text_buffer, "frequency: %d Hz",
tremolo_frequency);
                        draw_menu_line(text_buffer, 1);
                        sprintf(text_buffer, "depth: %d.%d", (int)tremolo_depth,
(int)(tremolo_depth*100));
                        draw_menu_line(text_buffer, 2);
                        SSD1306_DrawFilledRectangle(0, 7+10*3, 128, 10,
SSD1306_COLOR_BLACK);
                        SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                    }
                    else if(menu_page == 4)
                    {
                        chorus_init(&chorus, chorus_base_delay, chorus_depth,
chorus_lfo_freq, SAMPLING_FREQUENCY);
                        sprintf(text_buffer, "delay: %d ms",
(int)chorus_base_delay);
                        draw_menu_line(text_buffer, 1);
                        sprintf(text_buffer, "depth: %d ms", (int)chorus_depth);
                        draw_menu_line(text_buffer, 2);
                        sprintf(text_buffer, "LFO freq: %d Hz",
(int)chorus_lfo_freq);
                        draw_menu_line(text_buffer, 3);
                        SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                    }
                }
                else if(menu_screen == 2)
                {
                    if(menu_page == 0)
                    {
                        sprintf(text_buffer, "bass");
                        draw_menu_line(text_buffer, 1);
                        sprintf(text_buffer, "mid");
                        draw_menu_line(text_buffer, 2);
                        sprintf(text_buffer, "treble");
                        draw_menu_line(text_buffer, 3);
                        SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                    }

                    else if(menu_page == 1)
                    {
                        lowshelf_calculation(&lowshelf, 45000,
lowshelf_frequency, lowshelf_s, lowshelf_gain);
                        sprintf(text_buffer, "frequency: %d Hz",
lowshelf_frequency);
                        draw_menu_line(text_buffer, 1);
                        sprintf(text_buffer, "S = %d.%d", (int)lowshelf_s,
(int)(lowshelf_s*100));
                        draw_menu_line(text_buffer, 2);
                        sprintf(text_buffer, "gain: %d dB", lowshelf_gain);
                        draw_menu_line(text_buffer, 3);
```

```
                                SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                        }
                        else if(menu_page == 2)
                        {
                                peak_calculation(&peakingEQ, 45000, peak_frequency,
peak_q, peak_gain);
                                sprintf(text_buffer, "frequency: %d Hz", peak_frequency);
                                draw_menu_line(text_buffer, 1);
                                sprintf(text_buffer, "Q = %d.%d", (int)peak_q,
(int)(peak_q*100));
                                draw_menu_line(text_buffer, 2);
                                sprintf(text_buffer, "gain: %d dB", peak_gain);
                                draw_menu_line(text_buffer, 3);
                                SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                        }
                        else if(menu_page == 3)
                        {
                                highshelf_calculation(&lowshelf, 45000,
highshelf_frequency, highshelf_s, highshelf_gain);
                                sprintf(text_buffer, "frequency: %d Hz",
highshelf_frequency);
                                draw_menu_line(text_buffer, 1);
                                sprintf(text_buffer, "S = %d.%d", (int)highshelf_s,
(int)(highshelf_s*100));
                                draw_menu_line(text_buffer, 2);
                                sprintf(text_buffer, "gain: %d dB", highshelf_gain);
                                draw_menu_line(text_buffer, 3);
                                SSD1306_DrawFilledRectangle(0, 7+10*4, 128, 10,
SSD1306_COLOR_BLACK);
                        }
                }
                SSD1306_UpdateScreen();
                menu_redraw_flag = 0;
        }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();
  __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

  /** Initializes the RCC Oscillators according to the specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
```

```
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 8;
  RCC_OscInitStruct.PLL.PLLN = 180;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 2;
  RCC_OscInitStruct.PLL.PLLR = 2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Activate the Over-Drive mode
  */
  if (HAL_PWREx_EnableOverDrive() != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC1_Init(void)
{

  /* USER CODE BEGIN ADC1_Init 0 */

  /* USER CODE END ADC1_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC1_Init 1 */

  /* USER CODE END ADC1_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
number of conversion)
  */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
```

```
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel its corresponding rank in the
sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC1_Init 2 */

  /* USER CODE END ADC1_Init 2 */

}

/**
  * @brief ADC2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC2_Init(void)
{

  /* USER CODE BEGIN ADC2_Init 0 */

  /* USER CODE END ADC2_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC2_Init 1 */

  /* USER CODE END ADC2_Init 1 */

  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
number of conversion)
  */
  hadc2.Instance = ADC2;
  hadc2.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc2.Init.Resolution = ADC_RESOLUTION_12B;
  hadc2.Init.ScanConvMode = DISABLE;
  hadc2.Init.ContinuousConvMode = DISABLE;
  hadc2.Init.DiscontinuousConvMode = DISABLE;
  hadc2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc2.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO;
  hadc2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc2.Init.NbrOfConversion = 1;
  hadc2.Init.DMAContinuousRequests = ENABLE;
```

```
  hadc2.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  if (HAL_ADC_Init(&hadc2) != HAL_OK)
  {
    Error_Handler();
  }

  /** Configure for the selected ADC regular channel its corresponding rank in the
sequencer and its sample time.
  */
  sConfig.Channel = ADC_CHANNEL_1;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_15CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC2_Init 2 */

  /* USER CODE END ADC2_Init 2 */

}

/**
  * @brief DAC Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC_Init(void)
{

  /* USER CODE BEGIN DAC_Init 0 */

  /* USER CODE END DAC_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC_Init 1 */

  /* USER CODE END DAC_Init 1 */

  /** DAC Initialization
  */
  hdac.Instance = DAC;
  if (HAL_DAC_Init(&hdac) != HAL_OK)
  {
    Error_Handler();
  }

  /** DAC channel OUT1 config
  */
  sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC_Init 2 */

  /* USER CODE END DAC_Init 2 */

}
```

```c
/**
  * @brief I2C1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_I2C1_Init(void)
{

  /* USER CODE BEGIN I2C1_Init 0 */

  /* USER CODE END I2C1_Init 0 */

  /* USER CODE BEGIN I2C1_Init 1 */

  /* USER CODE END I2C1_Init 1 */
  hi2c1.Instance = I2C1;
  hi2c1.Init.ClockSpeed = 400000;
  hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
  hi2c1.Init.OwnAddress1 = 0;
  hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
  hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
  hi2c1.Init.OwnAddress2 = 0;
  hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
  hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
  if (HAL_I2C_Init(&hi2c1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN I2C1_Init 2 */

  /* USER CODE END I2C1_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 10-1;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 200-1;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
```

```
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 1000-1;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 9000-1;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */
```

```
}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA2_CLK_ENABLE();
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Stream5_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Stream5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Stream5_IRQn);
  /* DMA2_Stream0_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);
  /* DMA2_Stream2_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA2_Stream2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA2_Stream2_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  /*Configure GPIO pin : PC10 */
  GPIO_InitStruct.Pin = GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pins : PC11 PC12 */
  GPIO_InitStruct.Pin = GPIO_PIN_11|GPIO_PIN_12;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /* EXTI interrupt init*/
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}
```

```c
/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */
  __disable_irq();
  while (1)
  {
  }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```