Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

бакалавр
освітній равень

Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266

Назва теми

КвРКІ 20010. 20.01.01 ПЗ
Шифр

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Виконав: студент IV курсу, група КІ2ін-20-1 _____ Кахока Блессінг
Підпис Ініціали, прізвище

Керівник _____ С.М. Лисенко
Підпис, дата Ініціали, прізвище

Нормоконтролер _____ І.О. Засорнова
Підпис, дата Ініціали, прізвище

До захисту допускаю:
Зав. кафедри комп'ютерної інженерії та інформаційних систем _____ Т.О. Говорущенко
Підпис Ініціали, прізвище

«30» травня 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень бакалавр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

" 10 " 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Кахоці Блессінгу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266

Керівник проекту (роботи) Лисенко С.М., д.т.н., проф.

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 15.02.2024 р. № 8

2. Строк подання студентом проєкту (роботи) на кафедру 01.06.2024 р.

3. Вихідні дані до проекту (роботи) Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Системи підтримки мультимедіа на основі звукових кодеків

Опис компонентів програмно-технічного засобу управління звуковими кодеками на базі мікроконтролера esp8266

Реалізація програмно-технічного засобу управління звуковими кодеками на базі мікроконтролера esp8266

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Схеми пристрою

Блок-схема алгоритму

Блок-схема алгоритму

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Засорнова І.О., к.т.н., доцент		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання «_10_»_____01_____2024 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2024	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.02.2024	виконано
3	Робота над розділом 1 – дослідження предметної області та постановка задачі	01.03.2024	виконано
4	Робота над розділом 2 – вибір компонентів для проєктування системи адаптивного застосування моніторингових елементів розвідувального БПЛА	01.04.2024	виконано
5	Робота над розділом 3 – проєктування системи адаптивного застосування моніторингових елементів розвідувального БПЛА	29.04.2024	виконано
6	Оформлення пояснювальної записки згідно вимог	25.05.2024	виконано
7	Попередній захист ВКР	26.05.2024	виконано
8	Захист ВКР на засіданні ЕК	Червень 2024 року	

Студент _____ К. Блессінг
Підпис Ініціали, прізвище

Керівник роботи _____ С.М. Лисенко
Підпис Ініціали, прізвище

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 20010. 20.01.01 ПЗ	Пояснювальна записка	62		
			Графічні матеріали			
2		КвРКІ 20010. 20.01.01 Е8	Схеми пристрою	1		
3		КвРКІ 20010. 20.01.01 Е8	Блок-схема алгоритму	1		
4		КвРКІ 20010. 20.01.01 Е8	Блок-схема алгоритму	1		

Зм	Арк	№ докум	Підпис	Дата	КвРКІ 20010. 20.01.01 ВП		
Розробив		Блессінг			Відомість проекту	Літера	Аркуш 1 / Аркушів 1
Перевір.		Лисенко				У	
Н. контр.		Засорнова				ХНУ, КІ2ін-20-1	
Затв.		Говорущенко		12.06			

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266».

Автор роботи: Кахока Блессінг.

Керівник роботи: Лисенко Сергій Миколайович.

Пояснювальна записка: 62 с., 12 рис., 1 табл., 4 дод., 51 джерело.

Графічна частина: 3 креслення.

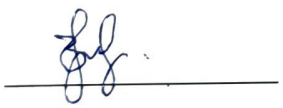
ПРОГРАМНО-ТЕХНІЧНИЙ ЗАСІБ, КЕРУВАННЯ ЗВУКОВИМИ КОДЕКАМИ, МІКРОКОНТРОЛЕР, ESP8266.

Метою дипломної роботи є розроблення програмно-технічного засобу керування звуковими кодеками на основі мікроконтролера ESP8266.

Об'єктом дослідження є керування звуковими кодеками на основі мікроконтролера ESP8266.

Предметом дослідження є програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266.

Під час проведення даного дослідження був використаний метод систематичного огляду літератури для вивчення і аналізу предметної області даного дослідження з текстових джерел інформації.



Підпис студента

10.06.2024

Дата

ЗМІСТ

ВСТУП 5

1 СИСТЕМИ ПІДТРИМКИ МУЛЬТИМЕДІА НА ОСНОВІ ЗВУКОВИХ КОДЕКІВ 7

1.1 Поняття звукових кодеків 7

1.2 Актуальність використання звукових кодеків 7

1.2.1 Кодування та декодування 8

1.2.2 Стиснення даних 8

1.2.3 Стиснення з втратами та без втрат (стиснення з втратами та без втрат) 8

1.2.4 Актуальність використання звукових кодеків 8

1.3 Процес стиснення даних в аудіокодеках 9

1.4 Типи кодеків 10

1.5 Відомі програмні системні бібліотеки для кодування та декодування аудіокодеків 12

1.6 Використання аудіокодеків в системах Інтернету речей (IoT) 14

1.7 Актуальність використання мікроконтролерів для систем Інтернету речей (IoT) 15

1.8 Керування звуковими кодеками на базі мікроконтролера ESP8266 17

1.9 Висновок 18

2 ОПИС КОМПОНЕНТІВ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ КЕРУВАННЯ ЗВУКОВИМИ КОДЕКАМИ НА БАЗІ МІКРОКОНТРОЛЕРА ESP8266 20

2.1 SP8266Audio 20

2.2 Встановлення 21

2.3 AudioInput 21

2.4 Класи AudioFileSource 22

2.5 Класи AudioGenerator 24

2.6 Класи AudioOutput 25
2.7 Adafruit I2S DAC і PCM5102 DAC 27
2.8 Програмне забезпечення I2S Delta-Sigma DAC (тобто відтворення музики з одним транзистором і динаміком) 28
2.9 Високочастотне гудіння за допомогою схеми 1-T 29
2.10 Налагодження схеми 1-T підсилювача 30
2.11 Оптичний вихід SPDIF 31
2.12 Використання зовнішньої пам'яті SPI для збільшення буфера 32
2.13 Використання SD-карт і ESP8266Audio 32
2.14 Портування на інші мікроконтролери 32
2.15 Висновок 33
3 РЕАЛІЗАЦІЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ КЕРУВАННЯ ЗВУКОВИМИ КОДЕКАМИ НА БАЗІ МІКРОКОНТРОЛЕРА ESP8266 35
3.1 Програмна реалізація 35
3.1.1 Реалізація заголовного файлу AudioFileSource 35
3.1.2 Реалізація заголовного файлу AudioFileSourceBuffer 39
3.1.3 Реалізація заголовного файлу AudioFileSourceBuffer 41
3.1.4 Реалізація заголовного файлу AudioFileSourceFATFS 42
3.1.5 Реалізація заголовного файлу AudioFileSourceHTTPStream 45
3.1.6 Реалізація заголовного файлу AudioFileSourceFS 47
3.1.7 AudioFileSourceFunction.cpp 49
3.1.8 Реалізація заголовного файлу функції AudioFileSourceFunction 52
3.1.9 AudioFileSourceHTTPStream.cpp 54
3.1.10 Реалізація заголовка AudioFileSourceHTTPStream 56
3.1.11 AudioFileSourceICYStream.cpp 59
3.1.12 AudioFileSourceICYStream.h 61
3.1.13 AudioFileSourceID3.cpp 64
3.1.14 AudioFileSourceID3 66
3.2 Висновок 68
ВИСНОВКИ 69
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 70

ДОДАТОК А БЛОК-СХЕМИ АЛГОРИТМІВ..... 75
ДОДАТОК Б КОПІЯ КРЕСЛЕННЯ «СХЕМИ ПРИСТРОЮ»..... 79
ДОДАТОК В КОПІЯ КРЕСЛЕННЯ «БЛОК-СХЕМИ АЛГОРИТМІВ» . 80
ДОДАТОК Г КОПІЯ КРЕСЛЕННЯ «БЛОК-СХЕМИ АЛГОРИТМІВ ... 81

ВСТУП

Актуальність системного програмного забезпечення (ПЗ) для підтримки мультимедіа в мікроконтролерних системах зростає в сучасному світі з кожним днем, і ось чому:

1. Збільшення функціональності мікроконтролерів. Сучасні мікроконтролери стають все більш потужними та функціональними. Вони вже не обмежуються простими завданнями, а використовуються в багатьох додатках, в тому числі і мультимедійних, таких як обробка аудіо, відео та графіки.

2. Розширення сфер застосування мікроконтролерів. Мікроконтролерні системи використовуються в широкому спектрі застосувань, включаючи автомобільну промисловість, медичні прилади, побутову техніку, промислові системи тощо. У багатьох з цих областей важливо мати можливість обробляти і відображати мультимедійний контент.

3. Зростаючі вимоги до візуалізації та інтерактивності. У багатьох випадках користувачі та системи потребують високоякісного звуку, зображень та інших мультимедійних елементів для забезпечення кращої взаємодії та візуалізації інформації.

4. Розвиток Інтернету речей (IoT). У мережі IoT велика кількість пристроїв потребує обробки та передачі мультимедійних даних. Наприклад, відеокамери, аудіосистеми, пристрої розумного будинку - всі вони потребують мультимедійної підтримки для ефективної роботи.

5. Віртуалізація та використання графічних інтерфейсів. Системи з графічним інтерфейсом потребують мультимедійної підтримки для ефективної віртуалізації та відображення графіки, що дозволяє створювати більш зручні та інтуїтивно зрозумілі пристрої.

6. Розширені можливості звукових систем. У багатьох додатках якість звуку має важливе значення, наприклад, в автомобільних системах, домашніх

аудіосистемах, медичних пристроях тощо. Підтримка мультимедіа є ключем до досягнення високої якості звуку.

Всі ці фактори роблять важливим розробку та впровадження системного програмного забезпечення для мікроконтролерів, що забезпечує підтримку мультимедіа, включаючи відеокодеки, графічні бібліотеки та інші компоненти для оптимальної обробки та відтворення мультимедійного контенту.

1 СИСТЕМИ ПІДТРИМКИ МУЛЬТИМЕДІА НА ОСНОВІ ЗВУКОВИХ КОДЕКІВ

1.1 Поняття звукових кодеків

Звуковий кодек (від «codec» - кодер-декодер) - це програмне або апаратне забезпечення, яке кодує і декодує аудіосигнали. Основне призначення аудіокодека - зменшити обсяг даних без суттєвої втрати якості звуку.

Важливо розуміти, що аудіосигнали в цифровому вигляді можуть займати набагато більше місця, ніж аналогові. Аудіокодеки використовуються для стиснення аудіоданих для більш ефективного зберігання та передачі. Їх можна розділити на дві основні категорії.

Кодеки з втратами (Lossy Codecs). Ці кодеки стискають аудіодані, видаляючи деяку інформацію, яка вважається менш важливою для людського вуха. Прикладами кодеків з втратами є MP3, AAC та Ogg Vorbis. Кодеки без втрат. Стискають аудіодані без втрати якості. Вони зменшують обсяг даних, але при цьому ви можете відновити оригінальний аудіосигнал без будь-яких втрат. Прикладами кодеків без втрат є FLAC, ALAC і WAV. Аудіокодеки використовуються в різних сферах, таких як аудіозапис, передача аудіо через мережі, потокове аудіо, а також у різних мультимедійних пристроях.

1.2 Актуальність використання звукових кодеків

Звуковий кодек - це програмне або апаратне забезпечення, яке може кодувати або декодувати аудіосигнали. Зазвичай його використовують для стиснення аудіоданих, щоб зменшити їхній розмір під час передачі або зберігання.

1.2.1 Кодування та декодування

Аудіокодек здатний перетворювати аудіосигнали в цифровий формат під час кодування і відновлювати їх у вихідну форму під час декодування. Цей процес важливий для забезпечення ефективного зберігання або передачі аудіоданих.

1.2.2 Стиснення даних

Однією з основних функцій аудіокодека є стиснення аудіоданих для зменшення розміру файлу або потоку даних. Це особливо важливо в сучасних мережах і пристроях, де обсяг даних є критичним фактором.

1.2.3 Стиснення з втратами та без втрат (стиснення з втратами та без втрат)

Деякі кодеки втрачають частину інформації під час стиснення (стиснення з втратами), що призводить до певної втрати якості звуку. Інші кодеки зберігають всю інформацію (стиснення без втрат), але можуть займати більше місця.

Важливо розуміти різні алгоритми стиснення, такі як MPEG, AAC, MP3, FLAC тощо. Кожен з цих алгоритмів має свої переваги та недоліки, які впливають на якість та обсяг стиснених аудіоданих.

1.2.4 Актуальність використання звукових кодеків

Актуальність використання звукових кодеків полягає в наступних сферах:

1. Мережева передача.
2. Зберігання мультимедійних даних.
3. Обробка звуку.
4. Мобільні пристрої.
5. Аудіо- та відеоконференції.

У сучасному світі, де стрімінгові сервіси, відеоконференції та онлайн-ігри є нормою, аудіокодеки відіграють ключову роль у передачі аудіоданих через інтернет. Ефективне стиснення дозволяє зберегти високу якість звуку при мінімізації пропускної здатності.

Аудіокодеки також використовуються для стиснення і зберігання аудіофайлів на пристроях і в хмарних сервісах.

У сфері обробки звуку та музичного виробництва звукові кодеки дозволяють працювати з аудіоданими, зменшуючи їх обсяг без значної втрати якості.

Звукові кодеки використовуються в смартфонах, навушниках та інших мобільних пристроях для оптимізації простору для зберігання і передачі аудіоданих.

Кодеки використовуються для забезпечення якісного звуку в платформах для відеоконференцій, де важливо мати чистий і реалістичний звук.

Розуміння аудіокодеків є важливим елементом, особливо якщо вони спрямовані на розробку програмного або апаратного забезпечення для обробки та передачі аудіоданих.

1.3 Процес стиснення даних в аудіокодеках

Стиснення даних в аудіокодеках - це процес зменшення обсягу аудіоінформації за допомогою різних методів, що дозволяє зберегти аудіофайл або аудіосигнал з меншою кількістю байт.

Це дозволяє зменшити пропускну здатність для передачі аудіоданих по мережах, заощадити місце на носіях і забезпечити ефективне використання ресурсів.

Аудіокодеки використовують два основні типи стиснення даних: з втратами та без втрат.

Під час стиснення з втратами частина аудіоінформації втрачається, щоб зменшити обсяг даних.

Цей метод широко використовується в таких аудіокодеках, як MP3, AAC, WMA та інших. Вони використовують різні алгоритми, такі як видалення непотрібних фрагментів, психоакустичні моделі та інші, щоб стискати аудіо без значної втрати якості для більшості слухачів.

Переваги методу: ефективний ступінь стиснення, добре підходить для стрімінгу та зберігання музики на пристроях з обмеженим обсягом пам'яті.

Недоліки методу: втрата якості, особливо при високому ступені стиснення. Але для більшості людей втрати непомітні або прийнятні.

Під час стиснення без втрат всі аудіодані зберігаються в точності і можуть бути відновлені в первісному вигляді.

Стиснення без втрат використовується в таких аудіокодеках, як FLAC, ALAC і APE. Це корисно в ситуаціях, коли важлива висока якість без втрати даних, наприклад, у музичній студії або архівному сховищі.

Переваги методу: Повністю відтворює оригінальну аудіоінформацію без втрати якості.

Недоліки методу: Файли без втрат зазвичай займають більше місця, ніж їхні аналоги з втратами, що робить їх менш практичними для зберігання на пристроях з обмеженою пам'яттю або для передачі через мережу з обмеженою пропускною здатністю.

Вибір між стисненням із втратами та без втрат залежить від конкретних потреб, які можуть включати якість звуку, обсяг пам'яті та швидкість передачі даних.

1.4 Типи кодеків

Існує багато типів аудіокодеків, кожен з яких має свої особливості та сфери застосування. Нижче наведено кілька типів аудіокодеків, які є популярними і широко використовуються.

MP3 (MPEG Audio Layer III). Тип: з втратами. Застосування: Один з найпоширеніших форматів для стиснення аудіо. Використовується для зберігання музики та аудіоконтенту на різних пристроях.

AAC (Advanced Audio Coding). Тип: з втратами.

Часто використовується в медіаплеєрах, смартфонах, потокових платформах, таких як Apple Music і Spotify.

FLAC (Free Lossless Audio Codec). Тип: без втрат. Забезпечує стиснення без втрат, часто використовується в аудіосистемах для аудіофілів і для зберігання архівних копій аудіофайлів.

ALAC (Apple Lossless Audio Codec). Тип: без втрат. Застосування: Розроблений компанією Apple, використовується в середовищах, де важлива сумісність з продуктами Apple.

WAV (Waveform Audio File Format). Тип: без стиснення (без втрат) або зі стисненням (з втратами).

Часто використовується для зберігання високоякісних нестиснутих аудіозаписів. Також може використовуватися з різними кодеками стиснення.

Opus. Тип: з втратами або без втрат. Оптимізовано для використання в режимі реального часу, наприклад, VoIP, потокового відео та аудіо чату.

Vorbis. Тип: з втратами. Безкоштовний кодек, який використовується у відеоіграх та інтернет-радіо. Dolby Digital (AC-3). Тип: з втратами. Широко використовується для кодування звуку у відеофайлах, DVD-дисках і на телебаченні. DTS (Digital Theater Systems). Тип: з втратами. Використовується в кінотеатрах, домашніх кінотеатрах і різних медіаплеєрах. Monkey's Audio (APE). Тип: без втрат. Використовується для стиснення аудіофайлів без втрати якості.

Вибір конкретного аудіокодека залежить від конкретних потреб, таких як якість, розмір файлу, підтримка пристроїв та інших факторів.

1.5 Відомі програмні системні бібліотеки для кодування та декодування аудіокодеків

Програмна бібліотека - це набір функцій і процедур, які можуть бути використані програмами для виконання певних завдань без необхідності писати код з нуля. Бібліотека містить готовий до використання код, який розробник може викликати у своїй програмі для виконання певних операцій.

Основні поняття, пов'язані з програмними бібліотеками:

1. Функції. Бібліотека містить функції, які можна викликати з програмного коду. Функції виконують певні завдання і дозволяють розробнику використовувати готовий код для виконання певних операцій.
2. Методи. Термін «метод» використовується, коли мова йде про об'єктно-орієнтовані бібліотеки. Метод - це функція, яка асоціюється з певним об'єктом або класом у програмі.
3. Класи та об'єкти. Деякі бібліотеки можуть включати класи та об'єкти, які надають структуру для організації коду та забезпечують об'єктно-орієнтований підхід до програмування.
4. Інтерфейси. Інтерфейси визначають спосіб взаємодії програми з бібліотекою. Якщо бібліотека має добре визначений інтерфейс, це полегшує її використання та інтеграцію в інші програми.
5. Документація. Бібліотека зазвичай супроводжується документацією, яка пояснює, як використовувати різні функції та методи. Документація важлива для розробників, які використовують бібліотеку.
6. Версії. Бібліотеки можуть мати різні версії, які включають виправлення помилок, нові функції та покращення. Розробники повинні вказувати, яку версію бібліотеки вони використовують.

Бібліотеки програмного забезпечення широко використовуються для полегшення процесу розробки програмного забезпечення та сприяння повторному

використанню коду. Вони можуть бути написані різними мовами програмування і надавати зручний інтерфейс для використання своїх функцій і методів.

Для кодування та декодування аудіокодеків доступні різні бібліотеки, залежно від мови програмування та платформи. Нижче наведено деякі з найпоширеніших бібліотек для розбору та декодування аудіокодеків:

1. FFmpeg. Мова: C. FFmpeg - це потужна бібліотека для обробки мультимедійних даних, яка підтримує декодування, кодування, перекодування та потокове передавання аудіо та відео. Підтримує широкий спектр аудіокодеків, що робить її популярним вибором для мультимедійних проектів.

2. Libsndfile. Мова: C. Libsndfile - бібліотека C, призначена для роботи з різними форматами аудіофайлів. Вона підтримує читання і запис аудіофайлів та надає простий інтерфейс для декодування і кодування аудіоданих.

3. Opus. Мова: C. Opus - це безкоштовний аудіокодек з відкритим вихідним кодом, призначений для ефективного стиснення та потокової передачі аудіо з низькою затримкою. Він підходить для широкого спектру застосувань і зазвичай використовується для передачі голосу через IP (VoIP) та спілкування в реальному часі.

4. Vorbis. Мова: C. Vorbis - це програма з відкритим вихідним кодом: Vorbis - це безкоштовний аудіокодек з відкритим вихідним кодом, розроблений фондом Xiph.Org. Він забезпечує високоякісне стиснення звуку і зазвичай використовується для потокового мовлення та інтернет-радіо.

5. MPG123. Мова: C. MPG123 - швидкий та вільно розповсюджуваний аудіокодек: MPG123 - це швидка, безкоштовна, з відкритим вихідним кодом бібліотека декодерів аудіо у форматі MPEG. Призначена для декодування аудіофайлів та потоків у форматі MPEG і широко використовується у різноманітних аудіопрограмах.

6. AudioToolbox. Мова: Objective-C (iOS/macOS). AudioToolbox - це середовище розробки від Apple для iOS та macOS. Він включає API для

декодування та кодування аудіофайлів, підтримуючи різні формати, такі як AAC, MP3 та інші.

7. JavaZoom JLayer. Мова: Java. JLayer - це Java-бібліотека для декодування MP3-файлів. Вона надає простий API для роботи з аудіо даними у форматі MP3 і зазвичай використовується в Java-додатках, які потребують функції декодування MP3.

8. GStreamer. Мова: C (з прив'язками для різних мов). GStreamer - це мультимедійний фреймворк, який підтримує побудову графів для обробки мультимедійних компонентів. До нього входять плагіни для різних кодеків, які дозволяють декодувати і кодувати аудіо та відео.

При виборі бібліотеки слід враховувати такі фактори, як мова програмування, яку ви використовуєте, конкретні аудіокодеки, які потрібно підтримувати, і ліцензійні вимоги для вашого проекту.

Також перевірте сумісність з цільовою платформою та рівень підтримки спільноти щодо поточного обслуговування та оновлень.

1.6 Використання аудіокодеків в системах Інтернету речей (IoT)

Використання аудіокодеків у системах Інтернету речей (IoT) може бути різноманітним і включати наступні аспекти:

1. Звукові сповіщення та тривоги. Аудіокодеки можна використовувати для відтворення звукових сигналів або сповіщень. Це може бути корисно для інформування користувачів про різні події або стан пристрою.

2. Голосові команди і відповіді. Вбудовані системи IoT можуть використовувати аудіокодеки для обробки і розпізнавання голосових команд користувача або для надання голосових відповідей.

3. Мультимедійні додатки. У деяких випадках, особливо в пристроях для відпочинку або розваг, аудіокодеки можуть використовуватися для відтворення музики або іншого мультимедійного контенту.

4. Телефонія та відеозв'язок. Деякі пристрої Інтернету речей можуть мати можливості телефонії або відеозв'язку, а аудіокодеки використовуються для обробки і передачі аудіоданих.

5. Аналіз звукових сигналів. У деяких випадках пристрої IoT можуть використовувати аудіокодеки для аналізу звукових сигналів, наприклад, для виявлення шуму, розпізнавання голосу або моніторингу звукового оточення.

6. Датчики гучності та звукові датчики. У деяких додатках аудіокодеки використовуються як чутливі датчики для вимірювання гучності або виявлення певних звукових патернів.

7. Системи моніторингу та безпеки. Застосування аудіокодеків може включати системи моніторингу та безпеки, де звук може бути використаний для виявлення надзвичайних ситуацій або подій.

8. Системи оцінювання мови. Аудіокодеки можуть бути використані для створення систем оцінювання мови, які аналізують якість і характеристики вимови.

Важливо враховувати, що вибір аудіокодеків в системах IoT пов'язаний з обмеженнями ресурсів, таких як живлення, пам'ять і обчислювальна потужність, що може вплинути на вибір того чи іншого рішення.

1.7 Актуальність використання мікроконтролерів для систем Інтернету речей (IoT)

Мікроконтролери залишаються ключовими компонентами для систем Інтернету речей (IoT) і є актуальними з кількох причин:

1. Низька вартість і невеликі розміри. Мікроконтролери характеризуються низькими виробничими витратами і малими габаритами, що робить їх ідеальними для вбудованих систем з обмеженими ресурсами, що характерно для багатьох пристроїв IoT.

2. Низьке споживання енергії. Багато мікроконтролерів оптимізовані для низького енергоспоживання, що є важливим аспектом для бездротових пристроїв IoT, які можуть працювати від батарей.

3. Вбудовані периферійні пристрої. Мікроконтролери часто мають вбудовані периферійні пристрої, такі як аналого-цифрові перетворювачі (АЦП), таймери і комунікаційні інтерфейси, що полегшує їх інтеграцію в системи IoT.

4. Підтримка бездротового зв'язку. Багато мікроконтролерів підтримують різні бездротові технології, такі як Wi-Fi, Bluetooth, Zigbee, LoRa, що робить їх ідеальними для підключення до мереж IoT.

5. Простота програмування. Багато мікроконтролерів мають добре розвинені засоби розробки та середовища програмування, які дозволяють розробникам легко створювати програми для їх використання.

6. Наявність широкого асортименту. Існує велика кількість мікроконтролерів різних виробників, що дозволяє розробникам вибрати пристрій, який найкраще відповідає їх конкретним потребам і вимогам проекту.

7. Спрощення розробки програмного забезпечення. Мікроконтролери зазвичай використовуються в простих вбудованих системах, що спрощує розробку програмного забезпечення для пристроїв IoT в порівнянні з більш потужними мікропроцесорами або мікрокомп'ютерами.

8. Можливості захисту та безпеки. Деякі мікроконтролери включають захист від атак і мають вбудовані інструменти для забезпечення безпеки пристроїв IoT.

9. Широкий спектр застосувань. Мікроконтролери використовуються в різних галузях, таких як автомобільна, медична, промислова, побутова техніка та інші, що доводить їх універсальність.

Таким чином, мікроконтролери залишаються актуальними для систем IoT завдяки своїм перевагам в оптимізації ресурсів, енергоефективності та підтримці бездротового зв'язку.

1.8 Керування звуковими кодеками на базі мікроконтролера ESP8266

Для реалізації управління аудіокодеками на базі мікроконтролера ESP8266 спочатку необхідно визначитись з конкретним аудіокодеком, який планується використовувати, оскільки різні кодеки можуть вимагати різних підходів.

Однак, загальний підхід до реалізації управління аудіокодеком може включати наступні кроки:

1. Вибір аудіокодека. Виберіть звуковий кодек, який відповідає вашим вимогам і можливостям ESP8266. Найпоширеніші кодеки включають PCM, I2S або аналоговий аудіовихід.

2. Підключення звукового кодека до ESP8266. Переконайтеся, що звуковий кодек правильно підключений до мікроконтролера ESP8266 за допомогою необхідних інтерфейсів (наприклад, I2S або SPI для цифрових кодеків, або аналогові контакти для аналогових кодеків).

3. Підключення та налаштування звуку. Використовуйте SDK або фреймворк для ESP8266 для налаштування підключення та конфігурації звуку. Це може включати встановлення правильних налаштувань (роздільна здатність, частота дискретизації тощо) для аудіовиходу.

4. Розробка програмного забезпечення для керування. Напишіть програмне забезпечення для ESP8266, яке дозволить вам керувати аудіовиходом. Це може бути код для ініціалізації, читання та відтворення аудіофайлів, відповіді на команди керування, аналізу аудіосигналів тощо.

5. Обробка звуку. Якщо необхідно, увімкніть функції обробки звуку, такі як еквалайзер, шумозаглушення або інші ефекти в залежності від вашого використання.

6. Тестування та налагодження. Протестуйте систему, переконайтеся, що звук виводиться коректно і відповідає вимогам. За потреби відрегулюйте налаштування.

7. Інтеграція в IoT-проект. Інтегруйте розроблене програмне забезпечення у ваш IoT-проект. Забезпечте взаємодію з іншими модулями та функціоналом пристрою.

8. Забезпечення безпеки та оптимізація. Враховуйте питання безпеки, зокрема, якщо пристрій підключений до інтернету. Оптимізуйте свій код для ефективного використання ресурсів ESP8266.

9. Документація та підтримка. Документуйте програмне забезпечення, яке ви розробляєте, і надавайте підтримку іншим розробникам, які можуть працювати з вашим кодом.

Цей процес може відрізнятися залежно від конкретного кодека, фреймворку або SDK, який ви використовуєте, але загальний підхід залишається однаковим для багатьох аудіо рішень на основі ESP8266.

1.9 Висновок

У цьому розділі було ретельно розглянуто декілька важливих моментів, щоб прояснити значення і потенційні переваги розробки системного програмного забезпечення для підтримки мультимедіа для мікроконтролерних систем на базі ATMega328.

Розробка системного програмного забезпечення (ПЗ) для підтримки мультимедіа на мікроконтролерних системах на базі ATMega328 може мати своє застосування та актуальність у ряді сценаріїв. Однак слід враховувати обмеженість ресурсів цих мікроконтролерів, зокрема в області пам'яті та обчислювальних можливостей. Нижче ми розглянемо деякі варіанти та сценарії, в яких розробка такої SDR може бути доречною:

Розробка SDR для обробки аудіоданих може бути актуальною для створення аудіопроектів, таких як прості синтезатори, запис аудіосигналів, звукові ефекти та інші аудіоподібні додатки.

Використання мікроконтролера для розробки простих музичних інструментів або секвенсорів, де важлива обробка аудіосигналів.

Підтримка візуалізації звуку за допомогою світлодіодів (наприклад, реакція на музику) може бути актуальною для розважальних та мистецьких інсталяцій.

У деяких випадках, коли необхідно обробляти або виводити на дисплей відеодані, можна розглянути розробку простих систем, наприклад, для відображення анімації або текстової інформації.

Розробка простих інтерактивних ігор або ігрових проектів, де важливими є звукові або графічні елементи.

Варто зазначити, що в сучасних умовах для складних завдань в області мультимедіа на мікроконтролерах часто використовуються більш потужні платформи, такі як ARM Cortex-M або застосування спеціалізованих чіпів, які мають більше ресурсів для таких завдань.

Таким чином, актуальність розробки системного програмного забезпечення для мультимедіа на мікроконтролерах ATMega328 залежить від конкретної задачі та вимог проекту, а також від готовності працювати з обмеженими ресурсами цього мікроконтролера.

2 ОПИС КОМПОНЕНТІВ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ КЕРУВАННЯ ЗВУКОВИМИ КОДЕКАМИ НА БАЗІ МІКРОКОНТРОЛЕРА ESP8266

2.1 SP8266Audio

Бібліотека Arduino, призначена для розбору і декодування аудіофайлів, таких як MOD, WAV, MP3, FLAC, MIDI, AAC і RTTL, полегшуючи відтворення на I2S ЦАП або програмно імітованому дельта-сигма ЦАП з динамічною 32x-128x передискретизацією.

Бібліотека пропонує повну підтримку ESP8266 з великою зрілістю, а також забезпечує значну підтримку ESP32, використовуючи як його вбудований ЦАП, так і зовнішні ЦАП.

Для автономного синтезу мови в реальному часі, розгляньте можливість використання ESP8266SAM. Ця бібліотека, яка базується на вищезгаданій бібліотеці, використовує стару програму синтезу на основі формантів для забезпечення мовних можливостей ESP8266, вимагаючи мінімального обсягу пам'яті і не потребуючи мережевого з'єднання.

Процедури MOD було отримано з StellarPlayer, а процедури MP3 - з libMAD.

Код декодування AAC взято з проекту Helix, ліцензованого під RSPL від RealNetwork. Для комерційного використання вам все одно знадобиться ліцензія AAC від Via Licensing.

ESP32 підтримує AAC-SBR, який багато веб-радіостанцій використовують для подальшого зменшення пропускної здатності. На жаль, ESP8266 не підтримує AAC-SBR через недостатню кількість вбудованої оперативної пам'яті.

Декодування MIDI досягається за допомогою сильно адаптованого MIDITONES у поєднанні зі значно оптимізованим за обсягом пам'яті шрифтом TinySoundFont. Більш детальну інформацію можна знайти у відповідних вихідних файлах.

Бібліотеки Opus, OGG та OpusFile взято з сайту Xiph.org, ліцензовано за ліцензією Xiph, з деталями патентів можна ознайомитися у src/{opusfile,libggg,libopus}/COPYING.

Декодування Opus наразі працює лише на ESP32 через його високі вимоги до пам'яті. Для проекту було використано бібліотеку ESP8266Audio.

Для початку ми повинні переконатися, що використовуємо версію 2.6.3 або новішу бібліотек Arduino для ESP8266, або останню версію ESP32 SDK від Espressif.

2.2 Встановлення

Встановіть бібліотеку в каталог

~/Arduino/libraries

mkdir -p ~/Arduino/libraries

cd ~/Arduino/libraries

git clone https://github.com/myP8266Audio

Перебуваючи в IDE, будь ласка, виберіть наступні опції на ESP8266:

Tools->lwIP Variant->v1.4 Open Source, або V2 Higher Bandwidth

Інструменти->Частота процесора->160МГц

2.3 AudioInput

Створіть джерело AudioInputXXX, яке вказуватиме на ваш вхідний файл, і приймач AudioOutputXXX, який можна налаштувати як I2S, I2S-sw-DAC або «SerialWAV» для запису WAV-файлу до послідовного порту для подальшого використання у вашій системі розробки.

Крім того, налаштуйте AudioGeneratorXXX для декодування вхідного сигналу і надсилання його на вихід.

Після цього вам потрібно викликати функцію AudioGeneratorXXX::loop() у вашому основному циклі loop() один або декілька разів.

Ця функція зчитує необхідні частини файлу, заповнює буфери I2S і негайно повертається.

Оскільки цей процес не керується перериваннями, довгі виклики delay() у вашому коді можуть спричинити затримки при відтворенні.

Щоб уникнути цього, або розбийте великі затримки на менші сегменти з викликами AudioGenerator::loop() між ними, або зменшіть частоту дискретизації, щоб зменшити кількість відліків, необхідних на секунду.

Зверніться до каталогу прикладів для базових демонстрацій.

2.4 Класи AudioFileSource

AudioFileSource: Це базовий клас, який реалізує простий «файловий» інтерфейс, доступний лише для читання. Його необхідність виникає через різноманітні реалізації файлових систем в Arduino, кожна з яких має свої власні унікальні атрибути.

Використовуючи цю обгортку, можна абстрагуватися від складнощів окремих файлових систем, тим самим спрощуючи AudioGenerator, який покладається виключно на ці прості функції.

AudioFileSourceSPIFFS призначений для читання файлів з файлової системи SPIFFS у контролері.

AudioFileSourcePROGMEM призначений для читання файлів з масиву PROGMEM. У UNIX-системах для отримання базового формату можна використати «xxd -i file.mp3 > file.h». Після цього необхідно додати «const» і «PROGMEM» до згенерованого масиву і включити його у скетч.

AudioFileSourceHTTPStream - це реалізація, що полегшує просте потокове читання HTTP для потокового MP3-трансляції типу ShoutCast.

Хоча він функціональний, він ще не повністю відмовостійкий. Більше того, при 44.1 кГц 128 біт він може заїкатися через обмеження процесора, хоча загалом працює за призначенням.

AudioFileSourceBuffer - Подвійна буферизація, корисна для HTTP-потоків.

AudioFileSourceBuffer слугує вхідним джерелом, яке доповнює вихід будь-якого іншого AudioFileSource додатковим буфером в оперативній пам'яті.

Ця функція особливо корисна для сценаріїв веб-потоків, де підтримка 1-2 пакетів у пам'яті має вирішальне значення для забезпечення плавного відтворення без переривань.

Щоб реалізувати цю функцію, спочатку створіть стандартне джерело вхідного файлу.

Потім створіть буфер з оригінальним джерелом на вході. Нарешті, передайте цей буферний об'єкт генератору.

Таке налаштування гарантує належну буферизацію аудіоданих, що покращує якість потокового передавання та мінімізує переривання відтворення.

```
AudioGeneratorMP3 *mp3;
AudioFileSourceHTTPStream *file;
AudioFileSourceBuffer *buff;
AudioOutputI2SNoDAC *out;
...
  // Створити HTTP потік звичайним чином
  file = new AudioFileSourceHTTPStream(«http://your.url.here/mp3»);
  // Створити буфер з цього потоку
  buff = new AudioFileSourceBuffer(file, 2048);
  out = new AudioOutputI2SNoDAC();
  mp3 = new AudioGeneratorMP3();
  // Передавати *буфер*, а не *http-потік*, щоб увімкнути буферизацію
  mp3->begin(buff, out);
...
```

AudioFileSourceID3 - фільтр парсеру потоку ID3 з користувацьким зворотнім викликом.

Цей клас приймає на вхід будь-який інший AudioFileSource і створює AudioFileSource, сумісний з будь-яким декодером.

Крім того, він автоматично витягує ID3-теги з MP3-файлів. Ви повинні визначити функцію зворотного виклику, яка буде викликатися після декодування тегів.

Це дозволить вам динамічно оновлювати стан інтерфейсу користувача (UI) цією інформацією. Для більш детальної інформації зверніться до прикладу PlayMP3FromSPIFFS.

2.5 Класи AudioGenerator

AudioGenerator є базовим класом для всіх декодерів файлів. Він потребує об'єктів AudioFileSource та AudioOutput для отримання даних та запису декодованих зразків відповідно. Щоб забезпечити безперервне відтворення без пропусків, викликайте його функцію loop() якомога частіше, щоб підтримувати буфери належним чином заповненими.

AudioGeneratorWAV - декодер, який читає і відтворює файли формату Microsoft WAVE (.WAV) з розрядністю 8 або 16 біт. AudioGeneratorMOD призначений для читання та відтворення файлів Amiga ModTracker (.MOD). Для обробки численних зчитувань SPIFFS, необхідних для отримання даних з необроблених зразків приладів, рекомендується використовувати тактову частоту 160 МГц. AudioGeneratorMP3 - програма для читання та відтворення файлів формату MP3 (.MP3) з використанням портованої бібліотеки libMAD.

Для плавного декодування MP3-файлів з бітрейтом 128 Кбіт 44.1 КГц рекомендується використовувати тактову частоту 160 МГц.

AudioGeneratorFLAC - декодер, що відтворює FLAC-файли за допомогою портованої бібліотеки libflac-1.3.2. Зазвичай йому потрібно близько 30 КБ пам'яті і мінімальний стек.

AudioGeneratorMIDI використовує синтезатор хвильових таблиць і вхід хвильових таблиць SoundFont2 для відтворення MIDI-файлів. Хоча теоретично він може обробляти до 16 одночасних нот, фактична кількість може змінюватися залежно від пам'яті, необхідної для структур SF2. AudioGeneratorAAC вимагає приблизно 30 КБ пам'яті і відтворює моно- або стерео AAC-файли за допомогою декодера AAC з фіксованою комою Helix. AudioGeneratorRTTTL пропонує насолоджуватися монофонічними 4-октавними мелодіями на вашому ESP8266 з дуже низькими вимогами до пам'яті та процесора для відтворення простих мелодій.

2.6 Класи AudioOutput

AudioOutput є базовим класом для всіх драйверів виводу. Він обробляє по одному зразку за раз і повертає true або false залежно від наявності вільного місця у буфері. Якщо повертається false, об'єкт, що викликав цей клас (AudioGenerator), відповідає за збереження даних, які не помістилися, і повторення спроби пізніше.

AudioOutputI2S є інтерфейсом і призначений для будь-якого I2S 16-бітного ЦАП. Він надсилає стерео або моносигнали на встановленій частоті. Протестовано з I2SDAC від Adafruit і ЦАП Beyond9032 з eBay, який надійно працює на частоті до 44,1 КГц. Щоб використовувати внутрішній ЦАП на ESP32, інстанціюйте цей клас як AudioOutputI2S(0,1).

AudioOutputI2SNoDAC - це клас, який використовує інтерфейс I2S для відтворення музики без ЦАП, ефективно перетворюючи її в дельта-сигма ЦАП з передискретизацією 32x або вище. Використовуйте надану схему для підключення динаміка або навушників до виводу I2STx (Rx). Залежно від використовуваного

транзистора, вам може знадобитися від'єднати вивід Rx від драйвера для послідовного завантаження. Цей інтерфейс підтримує лише моно-вихід.

AudioOutputSPDIF (експериментальний): Ще один інтерфейс, який використовує периферійні пристрої I2S для передачі бітового потоку S/PDIF, закодованого BMC. Для взаємодії з приймачем S/PDIF потрібен оптичний або коаксіальний приймач. Він повинен функціонувати навіть при базовому налаштуванні з червоним світлодіодом і струмообмежувальним резистором, підключеним до кабелю TOSLINK. Мінімальна підтримувана частота дискретизації - 32 КГц. Завдяки кодуванню BMC фактична швидкість передачі символів на виводі в 4 рази перевищує звичайну швидкість передачі даних I2S, що швидко виснажує буфери прямого доступу до пам'яті. AudioOutputSerialWAV: записує двійкові дані у форматі WAV із заголовками до послідовного порту. Якщо ви захопите послідовний вивід у файл, ви зможете відтворити його у вашій системі розробки.

AudioOutputSPIFFSWAV - клас, який записує двійкові дані у форматі WAV із заголовками до файлової системи SPIFFS. Перед викликом переконайтеся, що файлову систему змонтовано і запущено SPIFFS. Використовуйте функцію SetFilename() для вибору вихідного файлу перед запуском.

AudioOutputNull відкидає семпли до /dev/null. Переважно використовується для тестування швидкості, оскільки штучно не обмежує вихідну швидкість AudioGenerator, оскільки немає буферів для заповнення/зливання.I2S ЦАПи. У проекті використовується підсилювальний ЦАП Adafruit I2S +3W і звичайний ЦАП на базі PCM5102. Основні виводи представлені в таблиці 2.1.

Таблиця 2.1 - Основні виводи

I2S pin	Common label*	ESP8266 pin
LRC	D4	GPIO2
BCLK	D8	GPIO15
DIN	RX	GPIO3

Наведена тут колонка «загальне маркування» стосується широко розповсюджених плат для розробки NodeMCU та D1 Mini. Однак варто зазначити, що деякі виробники можуть використовувати різні схеми підключення, тому наведені тут позначення можуть не зовсім відповідати вашій конкретній моделі.

2.7 Adafruit I2S DAC і PCM5102 DAC

Це налаштування є простим і вимагає підключення лише контактів GND, VIN, LRC, BCLK і DIN. Ми повинні переконатися, що на вивід VIN подано +5 В, щоб досягти максимальної гучності. Було використано різні версії плат ЦАП PCM5102, і, незалежно від форм-фактора, всі вони мали однакову розводку. Окрім самого інтерфейсу I2S, на платі є декілька контактів для конфігурації входів, які необхідно підключити:

- Підключіть 3,3 В від ESP8266 до VCC, 33 В і XMT.
- Підключіть GND від ESP8266 до GND, FLT, DMP, FMT і SCL.
- Для стандартного інтерфейсу I2S:
- Підключіть BCLK до BCK.
- Підключіть I2SO до DIN.
- Підключіть LRCLK (WS) до LCK.

Існує багато варіантів, і більшість з них повинні адекватно працювати з цим кодом і ESP8266. Крім того, не забудьте під'єднати всі невикористані входи до GND або VCC, якщо це необхідно.

Залишення вхідного виводу плаваючим на будь-якій інтегральній схемі може призвести до нестабільної роботи, оскільки він може вловлювати шуми навколишнього середовища через дуже низьку вхідну ємність, що потенційно може спричинити збій у внутрішніх налаштуваннях мікросхеми.

2.8 Програмне забезпечення I2S Delta-Sigma DAC (тобто відтворення музики з одним транзистором і динаміком)

Для оптимальної якості звуку, особливо в стерео, рекомендується інвестувати в справжній I2S ЦАП. Adafruit пропонує чудовий моно варіант з підсилювачем, в той час як стерео ЦАП без підсилювача можна знайти недорого на eBay або інших платформах.

Однак, якщо ви шукаєте бюджетне рішення, програмний дельта-сигма ЦАП з 32-кратною передискретизацією (до 128-кратної, якщо дозволяє швидкість передачі аудіо) все одно забезпечить задовільну якість звуку. Для цього у вашому коді було використано об'єкт AudioOutputI2SNoDAC замість AudioOutputI2S. Крім того, зверніться до наступної схеми для керування динаміком потужністю 2-3 Вт за допомогою одного недорогого NPN транзистора 2N3904 і резистора ~1K 2N3904 (NPN). Схему пристрою показано на рисунку 2.1. Для покращення продуктивності також було включено конденсатор 220 мкФ між USB5V і GND. Це доповнення допомагає відфільтрувати будь-яке падіння напруги, що може виникнути під час відтворення великої гучності, сприяючи більш плавній роботі.

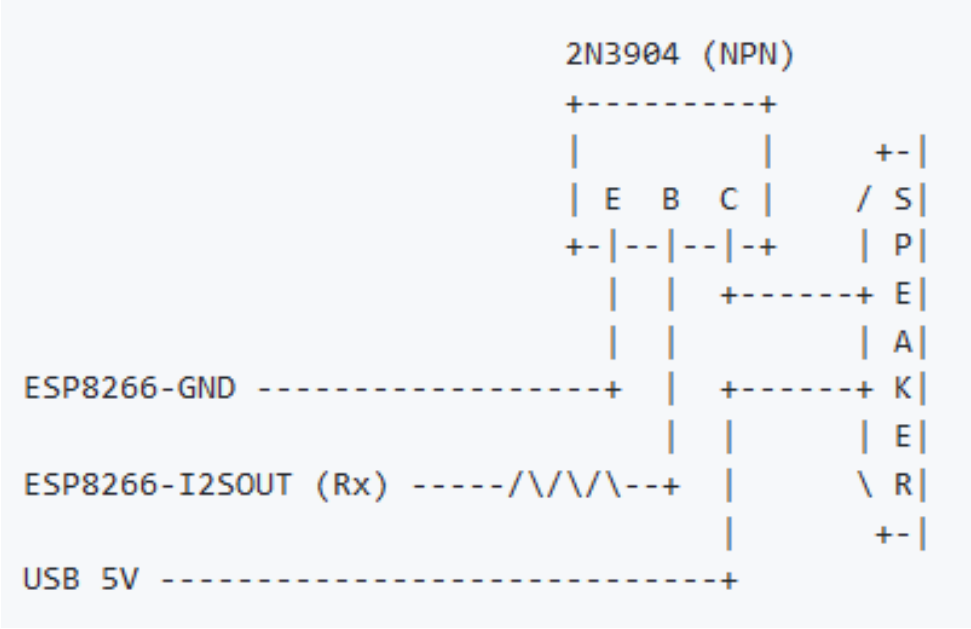


Рисунок 2.1 – Схема пристрою

Якщо джерело 5 В на вашій моделі ESP недоступне, ви можете використовувати 5 В від послідовного USB-адаптера або 3 В від ESP8266 (хоча це може призвести до зниження вихідної гучності).

Дуже важливо не намагатися керувати динаміком безпосередньо без транзистора. Виводи ESP8266 не можуть забезпечити достатній струм для ефективного управління навіть навушниками, і спроба зробити це може призвести до пошкодження вашого пристрою.

Ось підключення, як показано на рисунку 2.2.

```
ESP8266-RX(I2S tx) -- Resistor (~1K ohm, not critical) -- 2N3904 Base
ESP8266-GND        -- 2N3904 Emitter
USB-5V             -- Speaker + Terminal
2N3904-Collector   -- Speaker - Terminal
```

Рисунок 2.2 - Підключення

У попередній версії цієї схеми було пряме підключення ESP8266 до бази транзистора.

Хоча така схема забезпечує максимальну амплітуду, вона також може споживати більший струм від ESP8266, ніж безпечний, що потенційно може призвести до перегріву транзистора. В останньому випуску ESP8266Audio, з програмним дельта-сигма ЦАП, виводи LRCLK і BCLK тепер доступні для використання в додатках. Ви можете використовувати звичайні функції pinMode, digitalWrite або digitalRead за бажанням.

2.9 Високочастотне гудіння за допомогою схеми 1-T

Важливо зазначити, що підсилювач 1-T не може керувати будь-яким типом підсилювача. Якщо колонка потребує живлення, USB-входу, має підсвічування, Bluetooth або батарею, її не можна використовувати з цією схемою.

Вихід 1Т виробляє двійковий сигнал з напругою 0 або 5 В, без будь-яких проміжних рівнів. При підключенні безпосередньо до 8-омного паперового динаміка інерція дифузора діє як фільтр низьких частот, усереднюючи щільність імпульсів для отримання гладкого аналогового виходу. Однак, коли вихід 1Т підключено до підсилювача, він чергується між заземленням і перевантаженням входу операційного підсилювача на високій частоті. Це може викликати дзвін, а оскільки операційні підсилювачі зазвичай мають високочастотну характеристику, вони підсилюють високочастотний шум, що призводить до гудіння.

Та ж проблема може виникнути з п'єзодинаміками, які мають дуже високу частотну характеристику і практично не мають інерційності. Це може призвести до гудіння на високих частотах.

Одним з обхідних шляхів є підключення виходу 1Т до фільтру низьких частот, а потім подача сигналу на підсилювач.

Однак, на даний момент може бути простіше просто використовувати I2S ЦАП, який дозволяє уникнути цих проблем і забезпечує стерео і справжній 16-розрядний вихід.

2.10 Налагодження схеми 1-Т підсилювача

Якщо ви зібрали підсилювач, але не отримуєте жодного звуку, тут описано корисну послідовність налагодження для усунення проблеми.

Перевірте правильність підключення. Висновки GPIO і плати не завжди збігаються і можуть значно відрізнятися в різних марках плат ESP8266.

Ми повинні переконатися, що транзистор підключено правильно. Зверніться до специфікації конкретного типу корпусу і переконайтеся, що виводи під'єднані правильно. Зверніть увагу, що в цьому корпусі вивід, який знаходиться посередині однієї сторони, є колектором, а не базою, як можна було б очікувати.

Переконайтеся, що між колектором і емітером транзистора є приблизно 5 вольт. Подумайте, чи міг транзистор бути пошкоджений або перегрітий під час паяння або через неправильне підключення. За допомогою мультиметра перевірте

падіння напруги на діоді поза колом: від бази до емітера і від бази до колектора. Падіння напруги має становити від 0,5 до 0,7 вольт. Якщо він закорочений, відкритий або проводить струм в обох напрямках, замініть транзистор і переконайтеся, що він підключений правильно.

2.11 Оптичний вихід SPDIF

Рекомендується використовувати оптичний передавач TOSLINK, наприклад, TOTXxxx. Для тестування ви можете спробувати використати червоний світлодіод ~660 нм і резистор. Це налаштування подібне до вашого базового проекту Blink із зовнішнім світлодіодом, за винятком того, що світлодіод блиматиме дещо швидше (Малюнок 2.3).

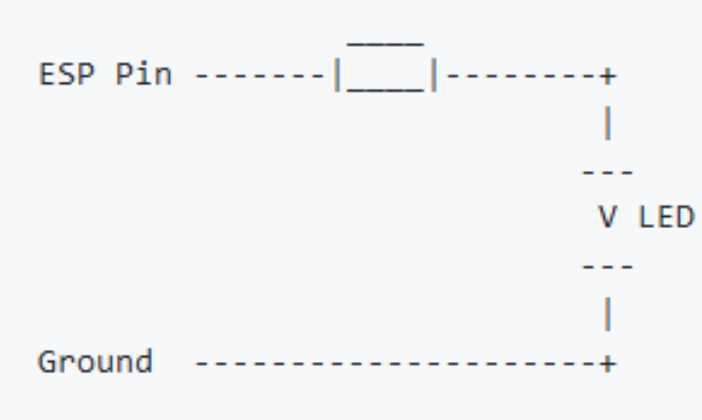


Рисунок 2.3 – Scheme for LED

Для ESP8266 з червоним світлодіодом (з прямим падінням напруги ~1,9 В) вам знадобиться мінімум 150-омний резистор (щоб обмежити струм максимум до 12 мА на вивід).

Вихідний вивід фіксований і повинен бути підключений до GPIO3/RX0.

На ESP32 вихідний вивід можна налаштувати за допомогою AudioOutputSPDIF(gpio_num), що забезпечує більшу гнучкість у виборі виводу.

2.12 Використання зовнішньої пам'яті SPI для збільшення буфера

Було реалізовано клас для полегшення використання 23LC1024 SPI RAM від Microchip як вхідного буфера. Ця мікросхема підключається до порту ESP8266 HSPI і забезпечує значний буфер для зменшення затримок при відтворенні потокового веб-контенту. Поточна версія дозволяє використовувати стандартний апаратний CS (GPIO15) або будь-який інший вивід за допомогою програмного забезпечення, хоча і з дещо зниженою продуктивністю. Нижче наведено приклад схеми, що ілюструє це налаштування (Рисунок 2.4).

2.13 Використання SD-карт і ESP8266Audio

Wemos SD card shield використовує GPIO15 як вибір мікросхеми SD. Це створює проблему, оскільки GPIO15 також призначено для I2SBCLK і він працює, навіть якщо ви використовуєте опцію NoDAC. Щоб вирішити цю проблему, нам потрібно змінити вибір мікросхеми (CS) на інший вивід і відповідно оновити вашу програму. Після цього налаштування ваша система повинна працювати правильно.

2.14 Портування на інші мікроконтролери

Процедури AudioGenerator не містять специфічного для ESP8266 коду, що робить перенесення на інші контролери відносно простим, за умови, що вони мають таку саму ендіанність, як і ядро Xtensa, що використовується. Якщо ми розглядаємо можливість портування на інший контролер, не соромтеся звертатися до мене. Можливо, я зможу запропонувати рекомендації, які допоможуть вам у правильному напрямку.

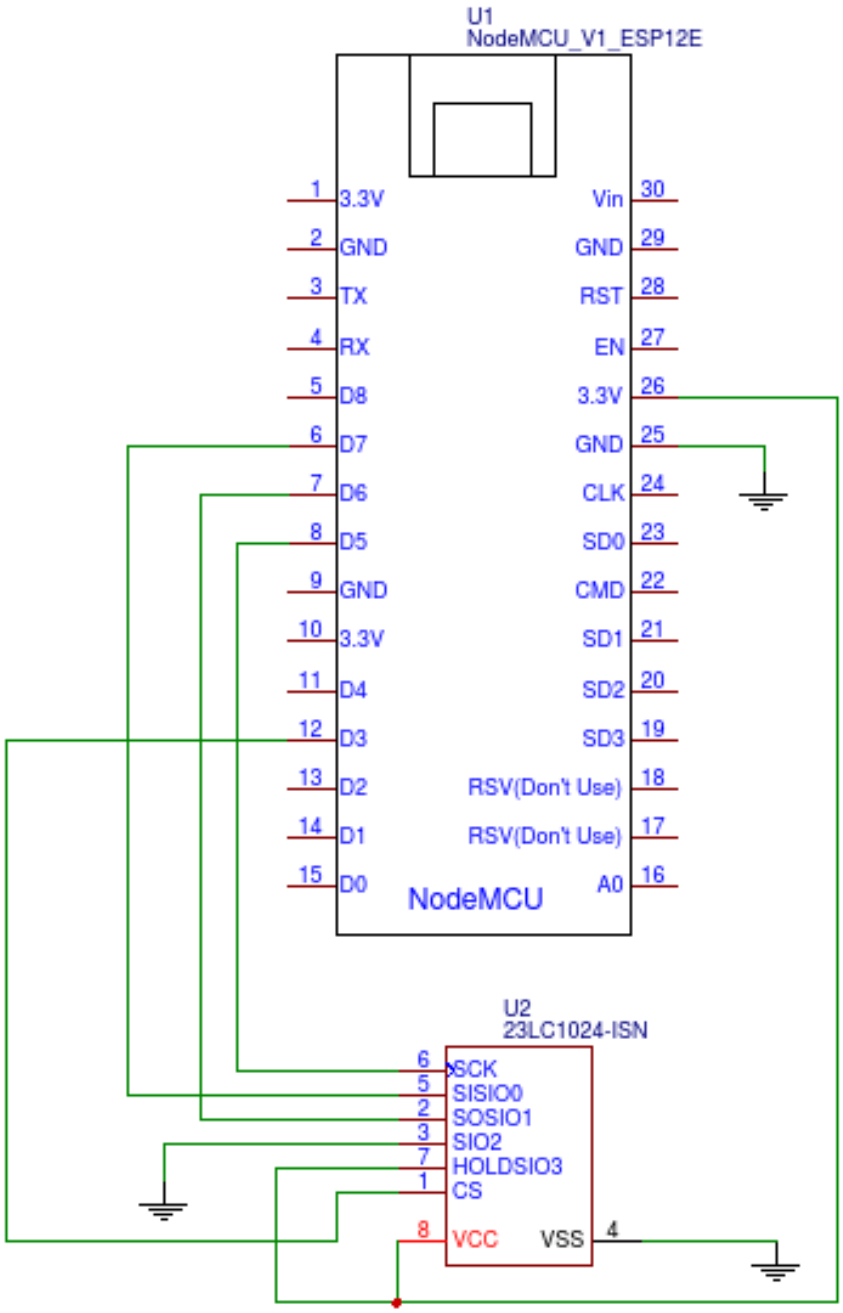


Рисунок 2.4 – Принципова схема

2.15 Висновок

У цьому розділі надано вичерпний огляд компонентів, програмного забезпечення та технічних засобів, необхідних для керування звуковими кодеками за допомогою мікроконтролера ESP8266. Розділ розпочався зі знайомства з бібліотекою ESP8266Audio, висвітлення її ролі та значення в управлінні

аудіоопераціями на платформі ESP8266. Надано детальні інструкції з налаштування середовища ESP8266Audio, щоб користувачі могли легко розпочати свої проекти. Було пояснено різні класи, такі як AudioInput, AudioFileSource, AudioGenerator і AudioOutput, з акцентом на їхніх функціональних можливостях і взаємодії для полегшення обробки аудіо. Було розглянуто інтеграцію та використання різних цифро-аналогових перетворювачів (ЦАП), включаючи Adafruit I2S DAC та PCM5102 DAC. Ці компоненти мають вирішальне значення для перетворення цифрових аудіосигналів в аналогові сигнали, які можна відтворювати через колонки. У розділі розглянуто реалізацію програмного I2S Delta-Sigma DAC, який дозволяє відтворювати музику, використовуючи мінімальне обладнання, а саме: один транзистор і динамік. Такий підхід демонструє універсальність та економічну ефективність мікроконтролера ESP8266 в аудіо додатках. Були надані практичні рішення для поширених проблем, таких як високочастотне гудіння за допомогою схеми 1-Т і налагодження схеми 1-Т підсилювача, що забезпечить більш плавний процес розробки і розгортання. Були обговорені такі складні теми, як оптичний вихід SPDIF і використання зовнішньої оперативної пам'яті SPI для збільшення розмірів буферів, що дозволило отримати уявлення про покращення звукових характеристик і розширення можливостей ESP8266. Було розглянуто використання SD-карт з ESP8266Audio, що дозволяє працювати з великими аудіофайлами і розширює ємність пам'яті мікроконтролера.

У розділі також розглянуто потенціал перенесення функцій ESP8266Audio на інші мікроконтролери, підкреслено адаптивність і масштабованість розглянутих методів. Розглядаючи ці ключові питання, глава надає розробникам знання та інструменти, необхідні для ефективного керування звуковими кодеками за допомогою мікроконтролера ESP8266. Універсальність і надійність платформи ESP8266 у поєднанні з детальним керівництвом відкривають шлях до створення інноваційних та ефективних аудіододатків.

3 РЕАЛІЗАЦІЯ ПРОГРАМНО-ТЕХНІЧНОГО ЗАСОБУ КЕРУВАННЯ ЗВУКОВИМИ КОДЕКАМИ НА БАЗІ МІКРОКОНТРОЛЕРА ESP8266

3.1 Програмна реалізація

Програмна реалізація включає в себе ряд заголовних файлів для реалізації програмно-технічного засобу управління звуковими кодеками на базі мікроконтролера esp8266. Розглянемо їх.

3.1.1 Реалізація заголовного файлу AudioFileSource

AudioFileSource - це базовий клас вхідного «файлу», який буде використовуватися AudioGenerator.

Наведений код визначає клас AudioFileSource, який слугує абстрактним базовим класом для керування джерелами аудіофайлів. Цей клас надає набір віртуальних функцій, які можуть бути реалізовані похідними класами для обробки різних операцій з аудіофайлами.

Ключові компоненти та функціональність класу AudioFileSource є наступними:

1. Конструктори та деструктори:

- AudioFileSource(): Конструктор за замовчуванням.
- ~AudioFileSource(): Віртуальний деструктор.

2. Операції з аудіофайлами:

- open(const char *filename) відкриває вказаний аудіофайл.
- read(void *data, uint32_t len) читає дані з відкритого аудіофайлу.
- readNonBlock(void *data, uint32_t len) читає дані з відкритого аудіофайлу у неблокуючий спосіб.
- seek(int32_t pos, int dir) шукає певну позицію у відкритому аудіофайлі.
- close() закриває відкритий аудіофайл.
- isOpen() перевіряє, чи наразі відкрито аудіофайл.

- getSize() повертає розмір відкритого аудіофайлу.

- getPos() повертає поточну позицію у відкритому аудіофайлі.

- loop() перевіряє, чи потрібно відтворювати аудіофайл у циклі.

3. Реєстрація зворотного виклику:

- RegisterMetadataCB(AudioStatus::metadataCBFn fn, void *data) реєструє функцію зворотного виклику для обробки подій метаданих аудіо.

- RegisterStatusCB(AudioStatus::statusCBFn fn, void *data) реєструє функцію зворотного виклику для обробки подій стану аудіо.

Клас AudioStatus використовується для керування метаданими та функціями зворотного виклику стану для класу AudioFileSource. Він надає методи для реєстрації функцій зворотного виклику та обробки відповідних подій.

У заголовному файлі представлено структуру та взаємозв'язки класу AudioFileSource і пов'язаних з ним компонентів. Діаграма містить детальні пояснення для кожного з методів класу, висвітлюючи їх призначення, параметри та значення, що повертаються.

Діаграма організована в єдину діаграму класів, де клас AudioFileSource є головним компонентом. Клас AudioStatus показано як окремий компонент, з відношенням композиції до класу AudioFileSource, що вказує на те, що клас AudioFileSource використовує клас AudioStatus для керування своїми зворотними викликами.

Примітки, додані до різних методів, надають стислі та інформативні пояснення, щоб допомогти користувачеві зрозуміти функціональність кожного компонента у класі AudioFileSource.

Ось детальний заголовний файл з поясненнями до наданого коду:Наданий код складається з декількох класів, які реалізують інтерфейс AudioFileSource, кожен з яких має власну функціональність та призначення.

Клас AudioFileSourceBuffer надає буферизований інтерфейс для читання даних з базового AudioFileSource. Він підтримує внутрішній буфер для оптимізації операцій читання та обробки потенційних умов неповного заповнення. Буфер може

бути виділений динамічно або наданий як попередньо виділений буфер пам'яті програмою.

Клас підтримує пошук у буфері та базовому AudioFileSource.

Він також надає методи для перевірки рівня заповнення буфера та зациклення базового AudioFileSource.

Клас AudioFileSourceFS надає реалізацію інтерфейсу AudioFileSource для читання файлів з файлової системи (наприклад, файлової системи SPIFFS або FAT).

Він інкапсулює специфічні для файлової системи операції, такі як відкриття, читання, пошук та закриття файлів.

Клас може бути ініціалізовано конкретним екземпляром файлової системи або ім'ям файлу для безпосереднього відкриття файлу.

AudioFileSourceFunction генерує аудіодані «на льоту» на основі наданих користувачем функцій зворотного виклику.

Вона встановлює WAV-заголовок із заданими параметрами звуку (тривалість, канали, частота дискретизації, розрядність) і генерує аудіодані за допомогою наданих функцій зворотного виклику.

Клас підтримує додавання однієї або декількох функцій зворотного виклику, як унікальних для кожного каналу, так і спільних для всіх каналів.

Він надає методи для читання згенерованих аудіоданих, пошуку в аудіо та отримання інформації про згенеровані аудіодані.

Клас AudioFileSourceHTTPStream надає реалізацію інтерфейсу AudioFileSource для читання аудіоданих з HTTP-потоку. Він інкапсулює функціональність HTTP-клієнта, включаючи відкриття потоку, зчитування даних та обробку розривів з'єднання. Клас підтримує повторне з'єднання з потоком у разі розриву з можливістю налаштування кількості повторних спроб і затримки. Він надає методи для читання даних з потоку, як у режимі блокування, так і без блокування, а також для отримання інформації про потік, наприклад, його розмір та поточну позицію. Файл заголовка надає детальне візуальне представлення логіки та функціональності кожного з цих класів. Підграфи представляють різні класи, а

вершини у кожному підграфі відповідають ключовим методам та операціям, що виконуються класом. Пояснення, надані поряд з елементами діаграми, допомагають зрозуміти призначення і поведінку кожного компонента коду, полегшуючи користувачеві розуміння загальної структури і функціональності реалізацій джерел аудіофайлів. Блок-схему алгоритму показано на рисунку 3.1.

Рисунок 3.1 - Блок-схема алгоритму

3.1.2 Реалізація заголовного файлу AudioFileSourceBuffer

У коді реалізовано клас AudioFileSourceBuffer, який виконує роль буферу для аудіофайлу, призначеного для відтворення. Він забезпечує ефективний доступ до аудіоданих, зменшуючи кількість зчитувань з основного джерела звуку.

Клас пропонує два конструктори для гнучкості:

Один приймає об'єкт AudioFileSource і розмір буфера як параметри. Це, ймовірно, створить новий буфер для зберігання аудіоданих.

Інший приймає об'єкт AudioFileSource, вказівник на наявний буфер і розмір буфера як параметри. Це дозволяє використовувати попередньо виділений буфер.

Клас містить деструктор для правильного звільнення пам'яті, виділеної для буфера.

Функції, що надаються класом, є:

- Seek дозволяє переміщувати позицію відтворення у аудіофайлі на основі різних режимів пошуку (наприклад, переміщення на певну кількість байт вперед або назад).

- Close закриває джерело звуку, ймовірно, звільняючи всі ресурси, пов'язані з ним.

- isOpen перевіряє, чи відкрито джерело звуку.

- getSize отримує розмір усього аудіофайлу.

- getPos отримує поточну позицію відтворення у аудіофайлі.

- getFillLevel повертає кількість байт, заповнених у буфері на даний момент. Це корисно, щоб визначити, скільки даних доступно для відтворення без необхідності зчитування з джерела.

- функція читання для доступу до даних. Для підвищення ефективності пріоритет віддається зчитуванню з буфера. Якщо у буфері недостатньо даних для задоволення запиту на читання, вона зчитує їх з основного джерела звуку.

- Функція заповнення проактивно намагається заповнити буфер даними з основного джерела звуку, коли буфер не заповнений. Це допомагає уникнути зависань під час відтворення, коли у буфері закінчуються дані.

- Зациклювати - увімкнути циклічне відтворення на основному джерелі звуку. Це корисно для створення безперервного відтворення аудіофайлу.

Блок-схема алгоритму показана на рисунку 3.2.

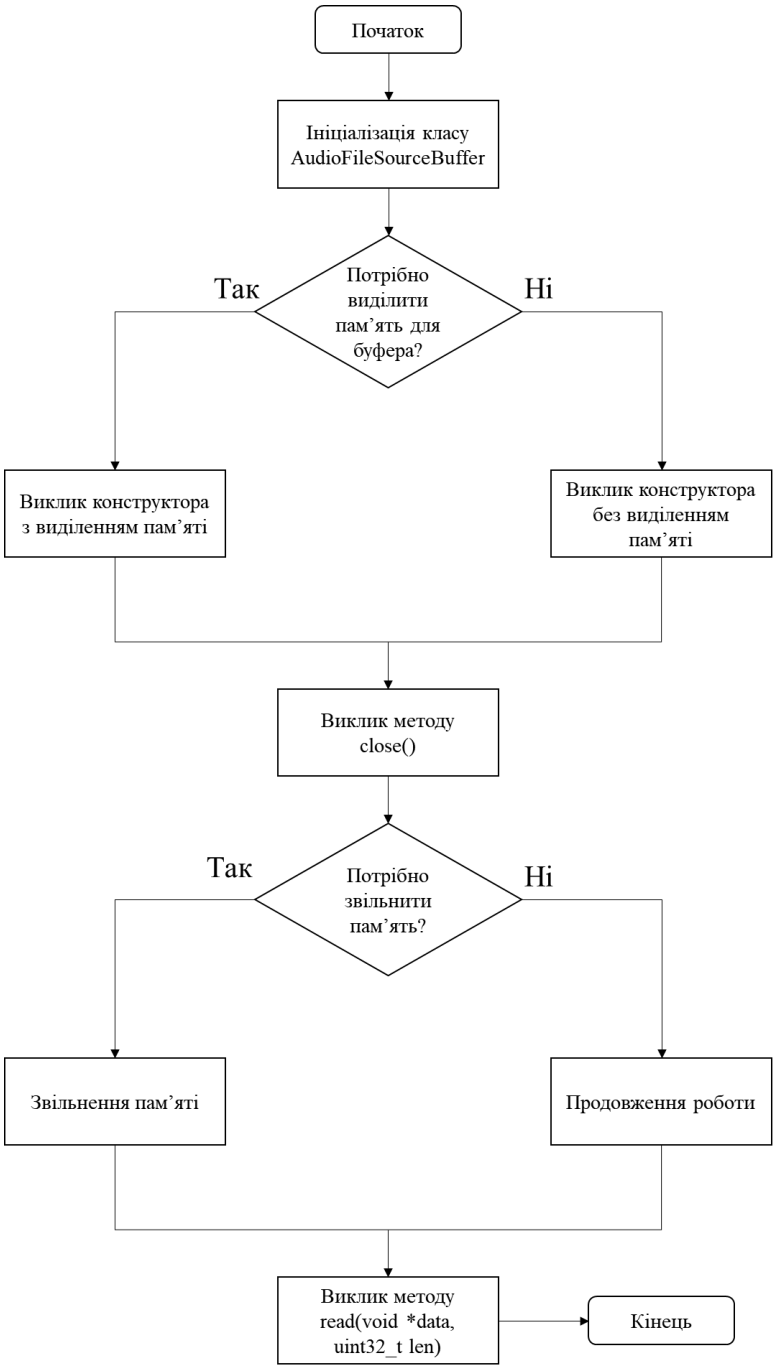


Рисунок 3.2 - Блок-схема алгоритму

3.1.3 Реалізація заголовного файлу AudioFileSourceBuffer

Цей код визначає заголовний файл для класу AudioFileSourceBuffer, який виконує роль буферу для аудіо даних під час відтворення. Заголовний файл виконує дві основні функції:

Конструктор отримує об'єкт AudioFileSource і розмір буфера як параметри. Це, ймовірно, створює новий буфер для зберігання аудіоданих.

Інший приймає об'єкт AudioFileSource, вказівник на наявний буфер і розмір буфера як параметри. Це дає змогу використовувати попередньо виділений буфер, що забезпечує гнучкіше керування пам'яттю.

Деструктор (~AudioFileSourceBuffer) відповідає за належне звільнення пам'яті, яку клас виділяє для буфера, забезпечуючи ефективне керування ресурсами.

Перевизначені віртуальні функції успадковуються від батьківського класу AudioFileSource і, ймовірно, перевизначаються для забезпечення функціональності буферизації навколо оригінальної реалізації.

Функція читання визначає пріоритет читання даних з буфера у першу чергу для підвищення ефективності. Якщо у буфері недостатньо даних для задоволення запиту на читання, вона зчитує дані з базового джерела звуку.

Шукати дозволяє вам переміщувати позицію відтворення у аудіофайлі на основі різних режимів пошуку (наприклад, переміщення на певну кількість байт вперед або назад).

Закрити - закриває джерело звуку, ймовірно, звільняючи всі ресурси, пов'язані з ним.

isOpen перевіряє, чи відкрито джерело звуку.

getSize отримує розмір усього аудіофайлу.

getPos отримує поточну позицію відтворення у аудіофайлі.

Loop дозволяє зациклити відтворення на базовому джерелі звуку. Це корисно для створення безперервного відтворення аудіофайлу.

Нова функція-член отримує кількість байт, заповнених у буфері на даний момент. Це корисно для визначення кількості даних, доступних для відтворення без необхідності зчитування з джерела, що підвищує ефективність.

Функцію fill оголошено приватною, що означає, що до неї може мати доступ лише сам клас AudioFileSourceBuffer. Ймовірно, вона відповідає за поповнення буфера з джерела, коли у буфері закінчуються дані, запобігаючи зупинкам відтворення.

Змінні-члени класу зберігають інформацію про буфер, джерело та стан відтворення. Src - це вказівник на об'єкт AudioFileSource, що представляє основне джерело звуку.

buffSize - розмір буфера у байтах.

Buffer - вказівник на комірку буферної пам'яті.

deallocateBuffer - прапорець, який вказує, чи відповідає клас за вивільнення буферної пам'яті.

writePtr - змінна, що відстежує позицію запису в буфері (куди записуються нові дані).

readPtr - змінна, що відстежує позицію читання в буфері (звідки зчитуються дані).

Length - кількість байт, заповнених у буфері на даний момент.

Filled - прапорець, що вказує на те, чи заповнений буфер на даний момент.

Інкапсулюючи ці дані у заголовному файлі, код сприяє кращій організації, зручності обслуговування та повторному використанню класу AudioFileSourceBuffer. Інші частини коду можуть взаємодіяти з класом через загальнодоступний інтерфейс без необхідності знати деталі внутрішньої реалізації.

3.1.4 Реалізація заголовного файлу AudioFileSourceFATFS

AudioFileSourceFS визначає заголовний файл для класу AudioFileSourceFATFS, спеціально розроблений для роботи з платами ESP32 для

відтворення аудіофайлів, що зберігаються у файловій системі FAT (зазвичай використовується у SD-картах). Він сприяє організації коду, багаторазовому використанню та розділенню завдань.

Клас успадковано від AudioFileSourceFS, що, ймовірно, забезпечує основу для обробки аудіофайлів з різних файлових систем. Це сприяє повторному використанню коду і зменшує надмірність, успадковуючи загальні функціональні можливості, такі як читання, пошук і закриття для маніпуляцій з аудіофайлами.

Перший конструктор (AudioFileSourceFATFS()) є конструктором за замовчуванням без аргументів. Ймовірно, він слугує базовим для створення об'єкта AudioFileSourceFATFS.

Другий конструктор (AudioFileSourceFATFS(const char *filename)) приймає ім'я файлу як параметр, але має іншу поведінку порівняно з типовим конструктором. Він ініціалізує батьківський клас (AudioFileSourceFS), але уникає виклику функції відкриття батьківського класу безпосередньо з назвою файлу. Це дає змогу уникнути непередбачуваної поведінки у реалізації відкритої функції батьківського класу.

Перевизначена функція open: Ця функція відіграє вирішальну роль у забезпеченні успішного відтворення з SD-картки.

Перевірка монтування файлової системи FAT (!FFat.begin()) перевіряє, чи файлову систему FAT вже змонтовано. Якщо ні, вона намагається змонтувати файлову систему за допомогою функції FFat.begin().

Обробка помилок (!FFat.begin()) записує повідомлення про помилку за допомогою audioLogger->printf_P, ймовірно, вказуючи на проблему з SD-карткою або файловою системою. Потім вона повертає false, сигналізуючи про помилку коду, що її викликає. Успішне монтування (FFat.begin()) викликає батьківський клас AudioFileSourceFS::open(ім'я файлу) для виконання власне процесу відкриття файлу за наданим ім'ям.

По суті, цей клас діє як адаптер для класу AudioFileSourceFS, спеціально пристосований для роботи з аудіофайлами, що зберігаються у файловій системі

FAT SD-карти на платах ESP32. Він забезпечує належне монтування файлової системи перед спробою відкрити аудіофайл, підвищуючи надійність і запобігаючи потенційним помилкам під час відтворення.

Блок-схему алгоритму показано на рисунку 3.3.

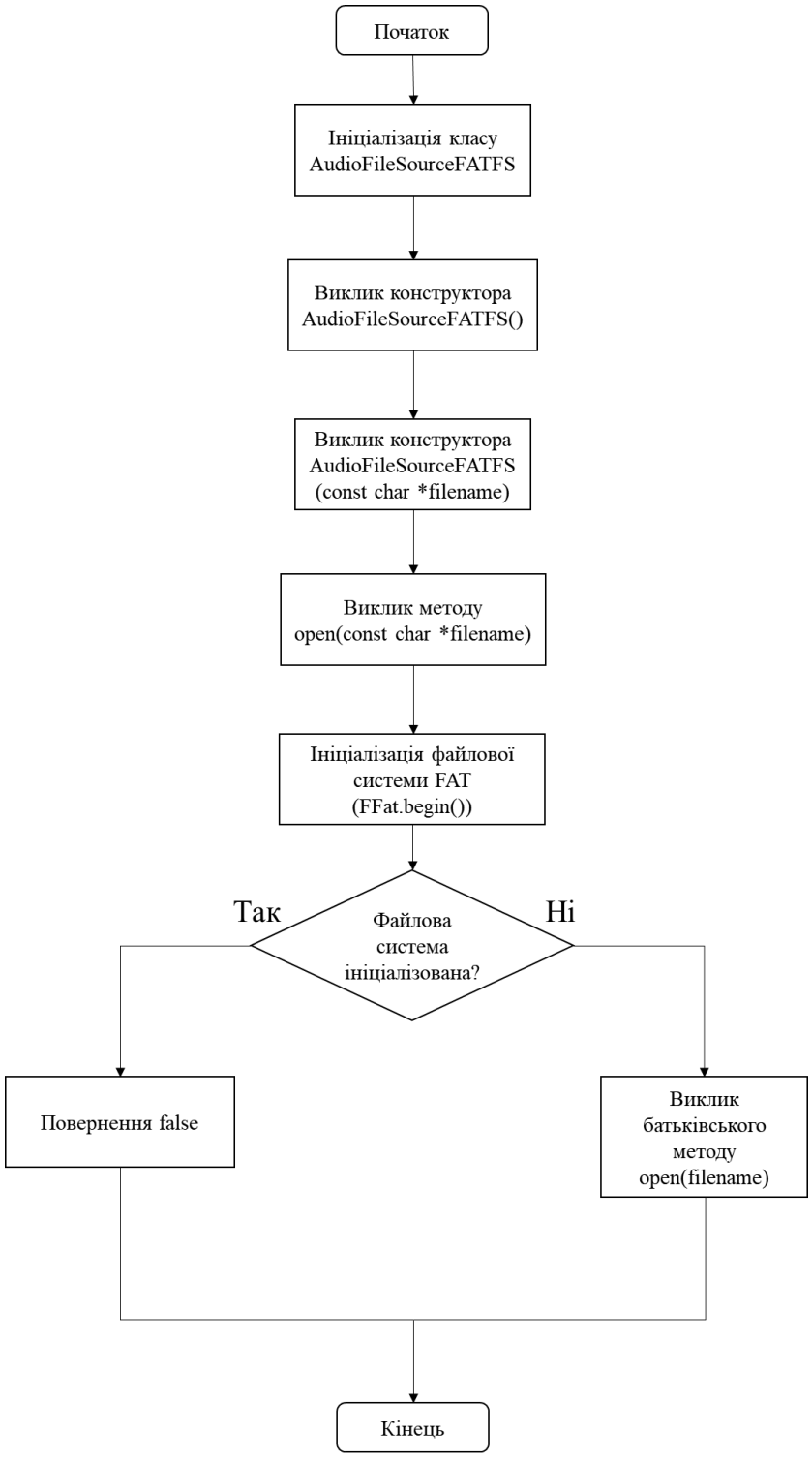


Рисунок 3.3 - Блок-схема алгоритму

3.1.5 Реалізація заголовного файлу AudioFileSourceHTTPStream

Заголовний файл AudioFileSourceHTTPStream визначає заголовний файл для класу AudioFileSourceHTTPStream, призначеного для відтворення аудіопотоків, що передаються за протоколами HTTP (наприклад, MP3) на платах ESP32 або ESP8266 з використанням фреймворку Arduino. Він сприяє модульності коду, повторному використанню та функціональності відтворення потоків.

1) Клас udioFileSourceHTTPStream успадковується від класу AudioFileSource, що, ймовірно, є основою для обробки аудіоданих у загальному вигляді. Це сприяє повторному використанню коду за рахунок успадкування таких функцій, як читання, пошук і закриття, які можна адаптувати до конкретного контексту HTTP-потоків. Успадковані функції можуть потребувати повторної реалізації для роботи з безперервними потоками, які не мають фіксованого розміру і традиційних можливостей пошуку.

2) Змінними-членами класу є:

3) - Client - об'єкт WiFiClient, ймовірно, використовується для керування низькорівневим мережевим з'єднанням з HTTP-сервером, на якому розміщується аудіопотік.

4) - http - об'єкт HTTPClient, використовується для надсилання HTTP-запитів та отримання фрагментів даних від сервера на більш високому рівні порівняно з WiFiClient. Це спрощує процес спілкування з сервером за допомогою протоколу HTTP.

5) - Pos - це цілочисельна змінна, яка може бути призначена для відстеження поточної позиції відтворення у потоці. Однак, на відміну від традиційних аудіофайлів з визначеним розміром, HTTP-потоки є безперервними, і пошук конкретних точок може бути неможливим. Значення pos можна використовувати для інформаційних цілей або для обмеження можливостей пошуку у потоці.

6) - Size - цілочисельна змінна, яка може зберігати повідомлений розмір потоку, якщо він доступний на сервері. Однак, на відміну від файлів, які можна

завантажити, HTTP-потоки зазвичай є відкритими, і їхній загальний розмір може бути невідомим заздалегідь. Значення розміру можна використовувати, якщо сервер надає достовірну інформацію про розмір.

7) - reconnectTries - ціле число, яке використовується для налаштування кількості спроб відновити з'єднання з сервером у разі втрати з'єднання під час відтворення. Це допомагає підвищити надійність у випадку перебоїв у мережі.

8) - reconnectDelayMs - ціле число, що визначає затримку (у мілісекундах) між спробами повторного з'єднання. Це дає змогу контролювати частоту повторних спроб і уникнути перевантаження сервера надто великою кількістю запитів на з'єднання за короткий час.

9) - saveURL - масив символів для зберігання URL-адреси аудіопотоку. Ймовірно, вона використовується для посилання на місцезнаходження потоку і потенційного керування кількома потоками або спробами перепідключення.

Конструктор FileSourceHTTPStream() призначений для створення порожнього об'єкта AudioFileSourceHTTPStream, потенційно ініціалізуючи змінні-члени значеннями за замовчуванням.

Конструктор AudioFileSourceHTTPStream(const char *url) отримує на вхід URL-адресу і може відповідати за декілька завдань:

Збереження URL-адреси у змінній-члені saveURL для подальшого використання.

Створення або ініціалізація об'єктів WiFiClient та HTTPClient для мережевої взаємодії.

Потенційне відкриття HTTP-з'єднання з наданою URL-адресою та ініціювання потоку. Залежно від реалізації сервера, це може включати надсилання певних HTTP-запитів або заголовків для початку потокової передачі аудіоданих.

Потенційно може буферизувати деякі початкові дані з потоку, щоб уникнути негайних затримок під час відтворення.

Деструктор (~AudioFileSourceHTTPStream) виконує належне очищення ресурсів, пов'язаних з мережевим з'єднанням та об'єктом HTTPClient. Він може

закрити з'єднання, звільнити будь-яку виділену пам'ять і забезпечити звільнення ресурсів для запобігання витоку пам'яті.

Перевизначені віртуальні функції забезпечують реалізацію, специфічну для обробки аудіопотоків HTTP, ймовірно, успадковуючи деяку поведінку від батьківського класу AudioFileSource, але адаптуючи її до особливостей безперервних потоків.

Функція open(const char *url) відповідає за відкриття HTTP-з'єднання за вказаною URL-адресою за допомогою об'єктів WiFiClient і HTTPClient. Вона може ініціювати процес отримання аудіоданих з сервера.

Функція read(void *data, uint32_t len) зчитує дані з потоку та заповнює наданий буфер аудіоданими. Вона може використовувати об'єкт HTTPClient для читання фрагментів даних з сервера у циклі, доки не буде заповнено запитувану кількість даних (len). Крім того, він може обробляти потенційні проблеми, такі як помилки з'єднання або недовикористання буфера (коли недостатньо даних для заповнення буфера). Тут може бути реалізовано логіку повторного з'єднання для автоматичної спроби відновити з'єднання, якщо його було втрачено під час відтворення.

Функція readNonBlock(void *data, uint32_t len) надає спосіб читання даних у неблокуючий спосіб, потенційно корисний для уникнення затримок у процесі відтворення.

3.1.6 Реалізація заголовного файлу AudioFileSourceFS

Заголовний файл AudioFileSourceFS було розроблено для роботи з різними файловими системами (FS) для відтворення аудіо на платах Arduino. Він забезпечує основу для роботи з аудіофайлами, що зберігаються на різних носіях, таких як SD-карти або внутрішня флеш-пам'ять. Це гарантує, що клас включається лише один раз у проект, запобігаючи помилкам від багаторазового включення.

Клас успадковується від AudioFileSource, який, ймовірно, є основою для обробки аудіоданих загалом. Це сприяє повторному використанню коду завдяки використанню наявних функцій для обробки помилок, базових операцій читання/запису та потенційних реалізацій базових класів для close та isOpen.

Конструктор AudioFileSourceFS(fs::FS &fs) отримує посилання на об'єкт fs::FS як параметр. Цей об'єкт, ймовірно, представляє конкретну файлову систему, що використовується (наприклад, файлову систему SD-карти, внутрішню файлову систему флешки).

Конструктор (AudioFileSourceFS(fs::FS &fs, const char *filename)) приймає як посилання на файлову систему, так і назву файлу як параметри. Ймовірно, він намагається відкрити вказаний аудіофайл за допомогою наданої файлової системи.

Перевизначені функції-члени надають функціональність для керування аудіофайлами у файловій системі. Функція open(const char *filename) відповідає за відкриття аудіофайлу, вказаного параметром filename, використовуючи файлову систему, на яку посилається змінна-член filesystem. Вона, ймовірно, виконує такі операції, як відкриття файлу, перевірка на наявність помилок і, можливо, зберігання хендла відкритого файлу у внутрішній пам'яті. read(void *data, uint32_t len) функція дозволяє читати аудіодані з відкритого файлу. Ймовірно, вона зчитує вказану кількість байт (len) у наданий буфер (data). Функція seek(int32_t pos, int dir) дозволяє здійснювати пошук у аудіофайлі на основі вказаної позиції (pos) та напрямку (dir). Реалізація може відрізнятися залежно від можливостей файлової системи. Функція close(), найімовірніше, закриває попередньо відкритий аудіофайл, звільняючи пов'язані з ним ресурси. Функція isOpen() перевіряє, чи файл наразі відкритий (змінна-член f), і повертає булеве значення, що вказує на стан відкритості. Функція getSize() може отримати розмір аудіофайлу з файлової системи, потенційно використовуючи функціональні можливості файлової системи для визначення розміру файлу. Функція getPos() отримує поточну позицію читання у відкритому файлі за допомогою методу f.position() об'єкта fs::File. Файлова система - це посилання на об'єкт fs::FS, що представляє конкретну файлову

систему, яка використовується. f: Об'єкт fs::File зберігає дескриптор відкритого аудіофайлу. Він використовується для подальших операцій читання/пошуку файлу. Інкапсулюючи ці функціональні можливості у заголовному файлі, код сприяє кращій організації, підтримці та повторному використанню класу AudioFileSourceFS. Інші частини коду можуть взаємодіяти з класом через загальнодоступний інтерфейс без необхідності знати низькорівневі деталі взаємодії з різними файловими системами. Блок-схему алгоритму показано на рисунку 3.4.

3.1.7 AudioFileSourceFunction.cpp

Генератор аудіовиходу може генерувати дані WAV-файлу з функції.

Цей код визначає клас з іменем AudioFileSourceFunction, який генерує аудіодані «на льоту» на основі наданих користувачем функцій і зберігає їх у форматі контейнера WAV. Він надає функції для керування згенерованими аудіоданими і пропонує інтерфейс, сумісний з класом AudioFileSource.

Конструктор (AudioFileSourceFunction) приймає аргументи, що визначають властивості аудіо.

Sec - тривалість аудіо у секундах (плаваюче число).

Channels - кількість каналів аудіо (наприклад, моно = 1, стерео = 2).

sample_per_sec - частота дискретизації (кількість відліків за секунду).

bits_per_sample - кількість біт на семпл (визначає якість звуку).

Він обчислює загальний розмір аудіоданих у байтах на основі наданих параметрів, а також створює структуру заголовка WAV (wav_header) і заповнює її відповідними значеннями розмірів фрагментів, деталями формату (канали, частота дискретизації тощо) на основі вхідних аргументів.

Розглянемо змінні-члени.

Funcs - вектор для зберігання наданих користувачем функцій (ймовірно, лямбд або вказівників на функції), які буде використано для генерування аудіоданих для кожного каналу.

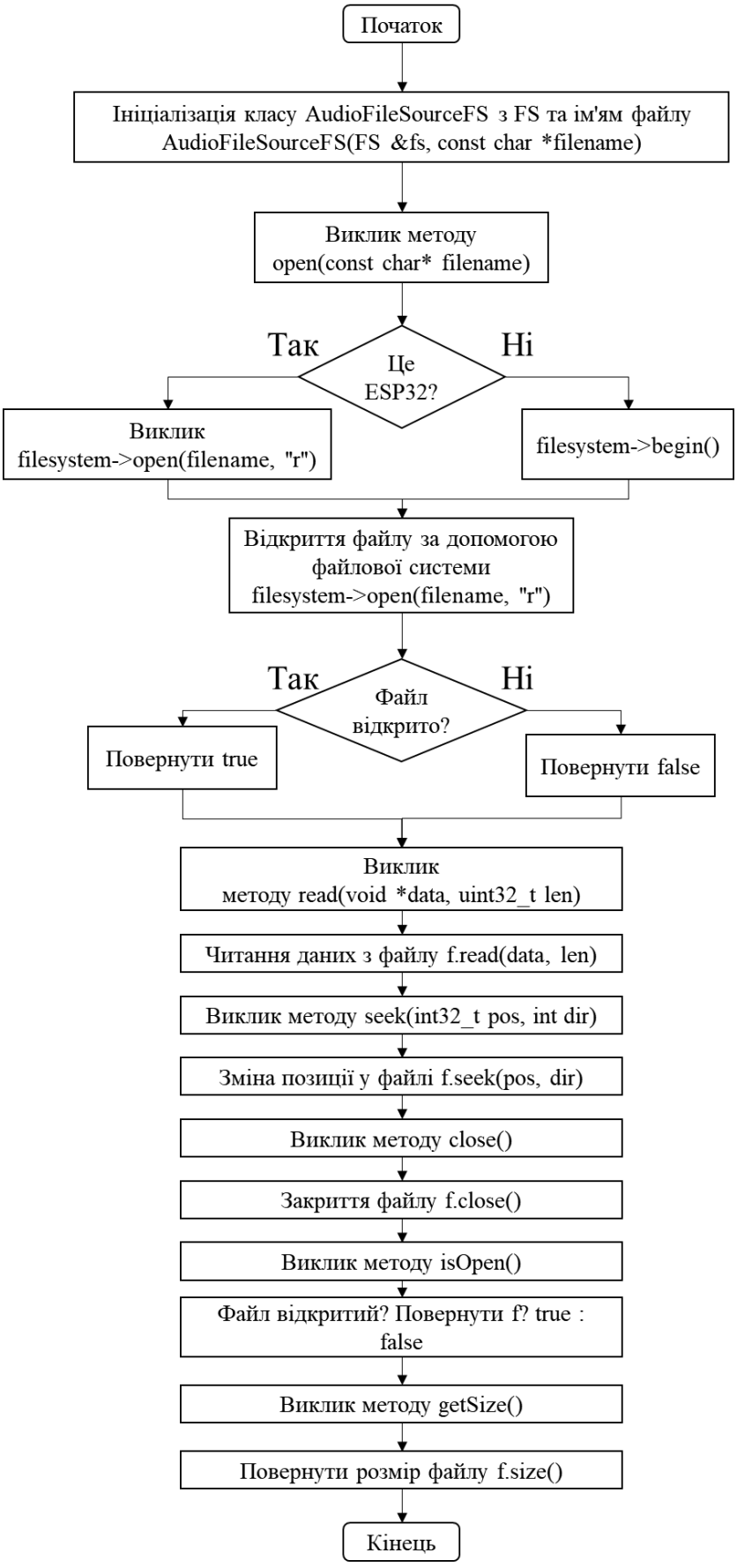


Рисунок 3.4 - Блок-схема алгоритму

Pos відстежує поточну позицію зчитування у аудіоданих.

Size зберігає загальний розмір аудіоданих (включно із заголовком).

is_ready - прапорець, який вказує, чи завершено генерацію аудіоданих і чи готові вони до читання.

is_unique - прапорець, який вказує, чи надані функції є унікальними для кожного каналу (інакше перша функція буде використана для всіх каналів).

Деструктор (~AudioFileSourceFunction) викликає функцію закриття для забезпечення належного очищення.

read(void* data, uint32_t len) зчитує аудіодані з WAV-контейнера у пам'яті та копіює їх у наданий буфер (data), дотримуючись заданої довжини (len).

Для заголовка він безпосередньо копіює байти з попередньо згенерованої структури wav_header.

Для аудіоданих обчислює поточний час на основі позиції зчитування і викликає відповідну функцію, надану користувачем (funcs[channel]), щоб згенерувати значення вибірки. Це робиться для кожного каналу.

На основі кількості бітів у вибірці перетворює значення вибірки з плаваючою комою у відповідне цілочисельне представлення і зберігає його у вихідному буфері.

Оновлює змінну pos для відстеження кількості прочитаних даних.

Повертає кількість фактично прочитаних байт, яка може бути меншою за запитувану довжину, якщо досягнуто кінця даних.

Функція seek(int32_t pos, int dir) дозволяє здійснювати пошук у згенерованих аудіоданих на основі вказаної позиції (pos) та режиму пошуку (dir).

SEEK_SET шукає з початку аудіоданих.

SEEK_CUR шукає відносно поточної позиції.

Виконує перевірку, щоб переконатися, що позиція пошуку знаходиться в межах допустимого діапазону аудіоданих.

Оновлює змінну-член pos для відображення нової позиції пошуку.

close() повертає змінні-члени до початкового стану, фактично закриваючи джерело звуку і звільняючи всі пов'язані з ним ресурси.

isOpen() повертає прапорець is_ready, який вказує, чи завершено генерацію аудіоданих.

getSize() повертає загальний розмір аудіоданих (включаючи заголовок).

getPos() повертає поточну позицію читання у аудіоданих.

Загалом, клас AudioFileSourceFunction надає спосіб генерування аудіоданих у пам'яті за допомогою визначених користувачем функцій і слугує віртуальним джерелом аудіофайлів, сумісним з класом AudioFileSource. Це дозволяє створювати динамічні звукові ефекти без необхідності використання попередньо записаних аудіофайлів.

Блок-схема алгоритму показана на рисунку А.1 Додатку А.

3.1.8 Реалізація заголовного файлу функції AudioFileSourceFunction

Генератор аудіовиходу може генерувати дані WAV-файлу з функції. Блок-схему алгоритму показано на рисунку А.2 Додатку А.

Клас AudioFileSourceFunction полегшує генерацію аудіоданих у пам'яті на основі наданих користувачем функцій і надає інтерфейс, сумісний з класом AudioFileSource. Це дає змогу створювати динамічні звукові ефекти «на льоту», не покладаючись на попередньо записані аудіофайли.

Структура заголовка WAV (WavHeader) вкладена в AudioFileSourceFunction, ця структура ретельно визначає макет заголовка WAV.

Вона використовує байт-масиви (масиви символів) для представлення різних полів заголовка, зокрема:

RIFF Chunk ідентифікує файл як контейнер WAV.

Блок формату визначає деталі аудіоформату (канали, частоту дискретизації, бітову глибину).

Блок даних вказує на розмір і розташування власне аудіоданих у файлі.

Об'єднання цілочисельних перетворень (Uint8AndIntX) (де X позначає 8, 16 або 32) забезпечують механізм ефективного перетворення між представленнями

цілих чисел зі знаком (intX) і беззнакових чисел (uintX) для різних розмірів даних (8, 16 або 32 біт).

Це виявляється корисним при пакуванні байт для зразків даних, забезпечуючи коректну маніпуляцію з числовими значеннями залежно від обраної розрядності.

Визначення вказівника функції (callback_t) діє як псевдонім для std::function<float(float)>. По суті, він являє собою вказівник на функцію або лямбда-вираз.

Змінними-членами є:

- std::vector<callback_t>, що зберігає функції генерації звуку, надані користувачем.

- Pos відстежує поточну позицію зчитування аудіоданих у пам'яті.

- Size зберігає загальний розмір згенерованих звукових даних, включаючи заголовок.

- is_ready - прапорець, який вказує, чи завершено процес генерації аудіоданих і чи готові вони до читання.

- is_unique - прапорець, який вказує, чи всі канали використовують ту саму функцію генерації звуку.

Конструктор (AudioFileSourceFunction) приймає аргументи, що визначають властивості аудіо, такі як тривалість (секунди), кількість каналів (моно-/стерео), частота дискретизації та розрядність на вибірку.

Ініціалізує структуру wav_header з наданими параметрами, ретельно налаштовуючи поля заголовка для точного відображення обраного аудіоформату.

Деструктор (~AudioFileSourceFunction) забезпечує належне керування ресурсами, викликаючи функцію закриття під час знищення об'єкта.

Функції addAudioGenerators слугують для додавання наданих користувачем функцій (однієї або декількох), які відповідатимуть за генерацію аудіо-зразків.

Шаблонна версія (addAudioGenerators(const F& f, Fs&&... fs)) дозволяє додавати декілька функцій з використанням варіативних аргументів, пропонуючи гнучкість у визначенні поведінки генерації звуку.

Вони виконують перевірку, щоб переконатися, що кількість наданих функцій відповідає кількості каналів або обмежується однією функцією (використовується для всіх каналів).

На основі результатів перевірки вони оновлюють прапори is_ready та is_unique, щоб відобразити дійсність конфігурації. Крім того, вони можуть очистити вектор функцій, якщо виявлено невірну конфігурацію.

Читання аудіоданих з WAV-контейнера у пам'яті.

Пошук у згенерованих аудіоданих на основі вказаної позиції та режиму пошуку. Закриття джерела звуку та звільнення пов'язаних з ним ресурсів.

Перевірка завершення генерації аудіоданих (готовність до зчитування).

Отримання загального розміру згенерованих аудіоданих (включно із заголовком).

Отримання поточної позиції читання в аудіоданих. По суті, клас AudioFileSourceFunction та його допоміжні функції дають змогу створювати динамічні звукові ефекти у вбудованих системах, таких як плати Arduino. Він надає механізм генерування аудіоданих у реальному часі за допомогою визначених користувачем функцій, усуваючи потребу у попередньо записаних аудіофайлах.

3.1.9 AudioFileSourceHTTPStream.cpp

Клас AudioFileSourceHTTPStream полегшує відтворення аудіопотоків, доставлених через HTTP на платах ESP32 та ESP8266. Він успадковується від класу AudioFileSource, дотримуючись спільного інтерфейсу для роботи з джерелами звуку. Цей клас дає змогу створювати функції відтворення аудіо, які використовують HTTP-потоки як джерело звуку, дозволяючи відтворювати

віддалений аудіоконтент без зберігання аудіоданих локально на пристрої. Блок-схема алгоритму показана на рисунку А.3 Додатку А.

HTTP-клієнт (http) - це екземпляр класу HTTPClient, який використовується для управління HTTP-з'єднанням з віддаленим сервером, на якому розміщується аудіопотік.

Він надає методи для відкриття з'єднань, надсилання запитів (як GET у цьому випадку), отримання відповідей та доступу до даних відповіді.

Клас реалізує механізми для встановлення та підтримки HTTP-з'єднання протягом усього процесу відтворення.

Він перевіряє код відповіді після надсилання GET-запиту, щоб переконатися в успішному з'єднанні (HTTP-код 200 - OK). Якщо виникає помилка, він закриває з'єднання, повідомляє про помилку за допомогою зворотного виклику (cb.st) і повертає код помилки.

Він включає логіку повторних спроб для обробки потенційних розривів з'єднання під час відтворення. Після розриву з'єднання вона намагається відновити з'єднання вказану кількість разів (reconnectTries) із затримкою між спробами (reconnectDelayMs). Інформативні повідомлення надаються через callback під час процесу перепідключення.

Функції read і readNonBlock надають механізми для отримання аудіоданих зі встановленого HTTP-потоку.

Вони обробляють потенційні ситуації, коли довжина запитуваних даних перевищує решту даних у потоці, коригуючи довжину зчитування, щоб уникнути виходу за межі потоку.

Функція readInternal слугує ядром для читання даних. Вона реалізує блокування або неблокування читання на основі встановленого прапора (nonBlock). Блокувальне читання очікує, поки дані стануть доступними за допомогою yield, тоді як неблокувальне читання перевіряє їхню доступність за допомогою stream->available().

Як тільки дані стають доступними, вони зчитуються з потоку за допомогою stream->read і зберігаються у наданому буфері.

Змінна-член pos оновлюється для відстеження кількості даних, прочитаних з потоку.

Функцію seek наразі не реалізовано (audioLogger->printf_P(...)), оскільки пошук у потоці HTTP може бути неможливим через природу потокового передавання даних. Вона завжди повертає false, щоб вказати, що пошук не підтримується.

Функція close граціозно розриває HTTP-з'єднання після завершення відтворення або у разі виникнення помилки.

Функція isOpen надає можливість перевірити, чи HTTP-з'єднання наразі активне.

Функція getSize, якщо сервер надає інформацію про довжину вмісту, повертає загальний розмір аудіопотоку. Однак для HTTP-потоків це може бути не завжди доступно.

Функція getPos повертає поточну позицію читання в потоці, вказуючи, скільки даних було прочитано на даний момент.

По суті, клас AudioFileSourceHTTPStream дає змогу розробляти додатки для відтворення аудіо на платах ESP32 і ESP8266, які використовують HTTP-потоки як джерело звуку. Він пропонує функції для керування з'єднанням, обробки помилок, читання даних та базових перевірок стану, що дозволяє створювати динамічні та віддалені можливості відтворення аудіо.

3.1.10 Реалізація заголовка AudioFileSourceHTTPStream

Клас AudioFileSourceHTTPStream полегшує відтворення аудіопотоків, доставлених через HTTP на платах ESP32 та ESP8266. Він успадковується від класу AudioFileSource, дотримуючись загального інтерфейсу для роботи з джерелами звуку у проекті. Цей клас дає змогу створювати функції відтворення аудіо, які

використовують HTTP-потоки як джерело звуку, що дозволяє відтворювати віддалений аудіоконтент без зберігання аудіоданих локально на пристрої. Блок-схема алгоритму показана на Рисунках А.4 та FA.5 Додатку А.

Код використовує умовну компіляцію (#ifdef та #else), щоб забезпечити включення відповідної бібліотеки HTTP-клієнта (HTTPClient для ESP32 або ESP8266HTTPClient для ESP8266) на основі цільової плати. Це сприяє перенесенню коду та його підтримці на різних платах для розробки.

Змінними-членами є

- client - об'єкт WiFiClient, ймовірно, використовується для встановлення базового мережевого з'єднання з віддаленим сервером, на якому розміщується аудіопотік.

- http - об'єкт HTTPClient, що використовується для керування HTTP-запитами (як GET у цьому випадку) та відповідями. Цей об'єкт обробляє деталі зв'язку з сервером.

- Відстеження позиції та розміру потоку:

- Pos відстежує поточну позицію читання в аудіопотоці, вказуючи, скільки даних було прочитано на даний момент. Це значення має вирішальне значення для відстеження прогресу відтворення.

- Розмір зберігає загальний розмір аудіопотоку, якщо він доступний у відповіді сервера. Це може бути не завжди доступно, особливо для прямих трансляцій.

- reconnectTries налаштовує кількість спроб повторного з'єднання, якщо з'єднання розривається під час відтворення. Цей механізм допомагає підвищити надійність у разі проблем з мережею.

- reconnectDelayMs задає затримку (у мілісекундах) між спробами перепідключення. Цей параметр дозволяє тонко налаштувати поведінку перепідключення, балансуючи між стійкістю та ефективністю.

- Зберегти URL - зберігає URL-адресу відкритого аудіопотоку для можливого повторного використання. Це може бути корисно у випадках, коли один і той самий

потік потрібно відтворювати кілька разів, або для керування різними сеансами відтворення.

AudioFileSourceHTTPStream() є конструктором за замовчуванням, який, ймовірно, ініціалізує змінні-члени значеннями за замовчуванням або порожніми станами.

AudioFileSourceHTTPStream(const char *url) приймає URL-адресу як аргумент. Він намагається відкрити аудіопотік за вказаною URL-адресою і встановити HTTP-з'єднання. Цей конструктор спрощує відкриття потоку для користувача.

Деструктор (~AudioFileSourceHTTPStream) забезпечує належне управління ресурсами, ймовірно, закриваючи HTTP-з'єднання при знищенні об'єкта. Це допомагає уникнути витоку ресурсів і потенційних проблем.

Функції управління потоком наступні:

- open(const char *url) відкриває вказану URL-адресу аудіопотоку і встановлює HTTP-з'єднання з віддаленим сервером. Ця функція відповідає за запуск процесу відтворення.

- close() граціозно завершує HTTP-з'єднання, звільняючи пов'язані з ним ресурси, такі як мережеві з'єднання. Ця функція необхідна для зупинки відтворення та очищення після використання.

- isOpen() перевіряє, чи активне наразі HTTP-з'єднання, вказуючи, чи відтворюється аудіопотік. Це може бути корисно для реалізації керування відтворенням або моніторингу стану відтворення.

Функції читання даних:

- read(void *data, uint32_t len) зчитує аудіодані з потоку у наданий буфер (data) з дотриманням запитуваної довжини (len). Ця функція реалізує поведінку блокування читання, тобто вона може зачекати, доки не буде доступною необхідна кількість даних, перш ніж повернутися.

- readNonBlock(void *data, uint32_t len) виконує неблокуюче читання, перевіряючи наявність даних перед продовженням. Ця функція корисна у

ситуаціях, коли важлива негайна відповідь, а очікування даних може призвести до затримок.

Статус та інформація:

- getSize() повертає загальний розмір аудіопотоку, якщо він доступний (може не завжди надаватися сервером). Це може бути корисно для оцінки тривалості відтворення або відображення інформації про прогрес.

- getPos() повертає поточну позицію зчитування у потоці. Це значення відображає, скільки даних було прочитано з потоку на даний момент, і може бути використане для відстеження прогресу відтворення.

3.1.11 AudioFileSourceICYStream.cpp

Код AudioFileSourceICYStream визначає клас AudioFileSourceICYStream, який успадковується від AudioFileSourceHTTPStream і полегшує відтворення аудіопотоків у форматі ICY (Icecast) на платах ESP32 і ESP8266. ICY - це потоковий протокол, який зазвичай використовується для інтернет-радіомовлення. Цей клас дозволяє розробляти додатки для відтворення аудіо, які використовують потоки ICY як джерело звуку.

Змінні-члени класу:

- icyMetaInt зберігає значення інтервалу метаданих, отримане із заголовка ICY, що вказує на частоту оновлення метаданих (наприклад, назви станції, інформації про пісню), вбудованих у потік. Це значення визначає кількість байт між блоками метаданих.

- icyByteCount відстежує кількість байт, прочитаних з моменту останнього зустрічного блоку метаданих ICY. Це значення має вирішальне значення для визначення потенційних перекриттів між запитуваними даними та межами метаданих під час процесу читання.

Загальнодоступними функціями-членами є:

- AudioFileSourceICYStream() - конструктор за замовчуванням, який, ймовірно, ініціалізує змінні-члени значеннями за замовчуванням або порожніми станами.

- AudioFileSourceICYStream(const char *url) - конструктор, який приймає URL як аргумент. Він намагається відкрити ICY-потік за вказаною URL-адресою, встановити HTTP-з'єднання з сервером і проаналізувати початкові заголовки для вилучення відповідної інформації.

- open(const char *url) відкриває вказану URL-адресу ICY-потоку за допомогою HTTP.

Вона надсилає додаткові заголовки в HTTP-запиті (Icy-MetaData: 1), щоб вказати серверу включити інформацію про метадані в потік.

Він збирає певні заголовки (icy-metaint, icy-name, icy-genre, icy-br) за допомогою функції collectHeaders з базової бібліотеки клієнта HTTP. Ці заголовки можуть містити назву станції, жанр, бітрейт і важливе значення icy-metaint, яке вказує на інтервал метаданих.

Аналізує отримані заголовки для вилучення значення icyMetaInt і потенційного заповнення назви станції, жанру та бітрейту (закоментовані секції // cb.md(...), ймовірно, використовують функцію зворотного виклику, щоб повідомити програму про цю інформацію). Зберігає довжину (розмір) контенту, якщо його надано сервером у заголовках відповіді.

Деструктор (~AudioFileSourceICYStream), успадкований від AudioFileSourceHTTPStream. Забезпечує належне управління ресурсами, ймовірно, закриваючи HTTP-з'єднання, коли об'єкт знищується.

Закрита функція-член readInternal(void *data, uint32_t len, bool nonBlock), яка перевизначає функцію readInternal з батьківського класу для обробки заголовків метаданих ICY. Логіка ядра зосереджена на обробці потенційних перекриттів між запитуваною довжиною даних (len) та межами метаданих ICY.

Це гарантує, що зчитування випадково не перетнеться з блоком метаданих ICY, що може порушити аудіопотік.

Якщо виявлено потенційне перекриття між запитуваними даними і межею метаданих:

Зчитує дані до межі метаданих, дотримуючись запитуваної довжини, але не виходячи за цю межу.

Він розбирає зустрінутий блок метаданих ICY для вилучення інформації на кшталт «StreamTitle» (закомментована секція // cb.md(...), ймовірно, з використанням функції зворотного виклику для сповіщення програми). Цей синтаксичний аналіз може включати читання додаткових байтів для визначення повного розміру блоку метаданих, а потім вилучення відповідних деталей.

Він пропускає решту байтів блоку метаданих, щоб уникнути передачі метаданих як частини аудіоданих.

Нарешті, він виконує фактичні дані, зчитані з потоку, використовуючи залишковий len після врахування потенційної обробки метаданих.

Він оновлює icyByteCount, щоб відстежити загальну кількість байт, прочитаних з моменту останнього зустрінутого блоку метаданих ICY. Це значення є важливим для точного обчислення потенційних перекриттів у наступних викликах читання.

По суті, клас AudioFileSourceICYStream розширює функціональність AudioFileSourceHTTPStream для обробки аудіопотоків у форматі ICY. Він включає логіку для розбору та обробки заголовків метаданих ICY, забезпечуючи при цьому безперебійне читання даних без переривання на межах метаданих. Це дозволяє розробляти надійні та інформативні додатки для відтворення потоків ICY на платах ESP32 та ESP866.

3.1.12 AudioFileSourceICYStream.h

Фрагмент коду AudioFileSourceICYStream.h розкриває оголошення заголовного файлу для класу AudioFileSourceICYStream, що дає змогу розробляти

програми відтворення аудіо, які обробляють потоки формату ICY (Icecast) на платах ESP32 та ESP8266. Давайте заглибимося в тонкощі цього коду:

<Arduino.h> надає фундаментальні функції для взаємодії з обладнанням та бібліотеками Arduino.

Умовні заголовки бібліотеки HTTP-клієнта такі:

- <HTTPClient.h> (для ESP32): Надає класи та функції для керування HTTP-запитами та відповідями на платі ESP32.

- <ESP8266HTTPClient.h> (для ESP8266) пропонує аналогічні функції, адаптовані до мережевих можливостей плати ESP8266.

- «AudioFileSourceHTTPStream.h» містить оголошення батьківського класу AudioFileSourceHTTPStream. Цей клас створює основу для обробки аудіопотоків, що передаються через HTTP, надаючи необхідні функції для керування з'єднанням, обробки помилок, зчитування даних та перевірки стану.

AudioFileSourceICYStream визначає клас з іменем AudioFileSourceICYStream, який успадковується від класу AudioFileSourceHTTPStream. Успадкування створює відношення батько-дочка, де дочірній клас (AudioFileSourceICYStream) успадковує функціональність батьківського класу (AudioFileSourceHTTPStream) і може додавати специфічну поведінку для обробки ICY-потоків.

Функції-члени є:

- AudioFileSourceICYStream() є конструктором за замовчуванням, який, ймовірно, відповідає за ініціалізацію змінних-членів значеннями за замовчуванням або порожніми станами.

- AudioFileSourceICYStream(const char *url) - конструктор, який приймає URL-адресу як аргумент. Ймовірно, він ініціалізує об'єкт і намагається відкрити ICY-потік за наданою URL-адресою, встановлюючи необхідне з'єднання і, можливо, розбираючи початкові заголовки.

- ~AudioFileSourceICYStream() є деструктором, який, ймовірно, викликається, коли об'єкт виходить за межі області видимості або явно

видаляється. Він необхідний для виконання завдань очищення, таких як закриття HTTP-з'єднання, пов'язаного з потоком, щоб уникнути витоку ресурсів і потенційних проблем.

- Функція uint32_t readInternal(void *data, uint32_t len, bool nonBlock) успадкована від батьківського класу (AudioFileSourceHTTPStream) і перевизначена у цьому дочірньому класі (AudioFileSourceICYStream). Це основний механізм для читання аудіоданих з потоку ICY. Ключове слово override вказує на те, що ця функція перевизначає поведінку однойменної функції батьківського класу, спеціалізуючи її на обробці тонкощів формату ICY, який включає в себе заголовки метаданих, що перемежовуються у потоці аудіоданих.

- int icyMetaInt зберігає значення інтервалу метаданих, отримане з заголовка ICY. Це значення визначає частоту (кількість байт) між оновленнями метаданих (наприклад, назви станції, інформації про пісню), вбудованих у потік.

- int icyByteCount

- відстежує кількість байт, прочитаних з моменту останньої зустрічі з блоком метаданих ICY. Він відіграє вирішальну роль у визначенні потенційних перекриттів між запитуваними даними та межами метаданих під час процесу читання. Підраховуючи прочитані байти, функція readInternal дозволяє уникнути випадкового зчитування блоків метаданих, забезпечуючи безперебійне відтворення аудіофайлу.

По суті, заголовний файл AudioFileSourceICYStream.h закладає основу для класу, спеціально призначеного для обробки аудіопотоків у форматі ICY. Він успадковує функції класу AudioFileSourceHTTPStream для загального управління HTTP-потоками і додає спеціалізовану поведінку для обробки заголовків метаданих ICY, що дозволяє розробникам створювати надійні та інформативні програми відтворення ICY-потоків на платах ESP32 і ESP8266.

3.1.13 AudioFileSourceID3.cpp

Цей заголовний файл представляє клас AudioFileSourceID3, призначений для легкої обробки метаданих ID3 у аудіофайлах або потоках. Він успадковується від класу AudioFileSource, який, ймовірно, надає функції для читання аудіоданих з різних джерел. Ось розбивка ключових елементів, що заглиблює у внутрішню роботу:

AudioFileSourceID3 успадковується від AudioFileSource. Це встановлює зв'язок між батьком і дитиною, де AudioFileSourceID3 успадковує основні функції для читання аудіоданих від батьківського класу (AudioFileSource). Потім він додає специфічну поведінку для обробки розбору метаданих ID3, що робить його спеціалізованим класом, пристосованим для цієї мети.

Змінними-членами є:

- src - вказівник, що зберігає посилання на базовий об'єкт AudioFileSource. Клас AudioFileSourceID3 діє як обгортка, делегуючи більшість завдань з читання базовому джерелу, а сам займається розбором метаданих у форматі ID3. По суті, він перехоплює потік даних з джерела, розбирає ID3-теги на початку (якщо вони присутні), а потім дозволяє решті аудіоданих пройти до програми.

- Перевірено - це булевий прапорець, який виконує роль воротаря, гарантуючи, що ID3-заголовок буде перевірено і розібрано лише один раз. Цей прапорець запобігає надлишковій обробці при наступних читаннях.

- AudioFileSourceID3(AudioFileSource *src) ініціалізує об'єкт посиланням на базовий об'єкт AudioFileSource. Це створює екземпляр класу AudioFileSourceID3, готовий до використання для читання аудіоданих під час обробки метаданих ID3.

- ~AudioFileSourceID3() виконує всі необхідні завдання з очищення, пов'язані з об'єктом. У реальних сценаріях це може включати закриття будь-яких тимчасових ресурсів, використаних під час розбору ID3, або вивільнення пам'яті.

- Функція read(void *data, uint32_t len) відповідає за читання даних. Вона діє як точка входу для запитів даних програми. Ось покрокова розбивка її логіки.

- Початковий Checkst перевіряє прапорець checked.

Якщо прапорець дорівнює false (що означає, що заголовок ID3 ще не було оброблено), він зчитує початкові 10 байт з базового джерела, щоб перевірити наявність підпису заголовка ID3 (наприклад, «ID3»).

Якщо підпис відповідає формату ID3, він аналізує заголовок, щоб витягти таку інформацію, як версія ID3, розмір, наявність байтів несинхронізації та розширеного заголовка. Цей розбір допомагає визначити, як інтерпретувати наступні дані. Він створює об'єкт AudioFileSourceUnsync (описано далі) для обробки несинхронізації, якщо це необхідно. Байти несинхронізації використовуються у деяких версіях ID3 для запобігання неправильним інтерпретаціям під час синтаксичного аналізу.

Пропускає розширений заголовок (якщо він є), оскільки він не є важливим для вилучення основних метаданих, таких як альбом, назва або виконавець.

Входить у цикл для перебору потенційних кадрів ID3, доки не буде досягнуто кінця даних ID3:

Зчитує ідентифікатор кадру (зазвичай 4 байти), щоб визначити тип інформації, яку містить кадр (наприклад, назва альбому, ім'я виконавця).

Зчитується розмір кадру, щоб визначити, скільки байт відведено на вміст цього конкретного кадру.

Перевіряє стиснення кадру (наразі не підтримується наданим кодом).

Читає вміст кадру на основі розміру кадру і прапорця несинхронізації. Це може включати читання додаткових байтів, якщо увімкнено несинхронізацію.

Витягує відповідну інформацію, таку як альбом, назва, виконавець, рік тощо, на основі ідентифікатора кадру. Потенційно він може викликати функцію зворотного виклику (cb.md), щоб повідомити програму про знайдені метадані. Ця функція зворотного виклику, ймовірно, знаходиться в окремій частині коду і відповідає за осмислену обробку видобутих метаданих (наприклад, відображення їх у користувацькому інтерфейсі).Once all ID3 frames are processed, it signals the end

of ID3 tags using the callback function (cb.md), informing the application that the following data is the actual audio content.

Після початкової перевірки ID3 (або якщо не знайдено жодного заголовка ID3), він делегує фактичне читання даних базовому джерелу (src->read) для решти даних (аудіоконтенту). Таким чином, по суті, ми обходимо логіку розбору ID3 і отримуємо необроблені аудіобайти з джерела.

3.1.14 AudioFileSourceID3

Заголовний файл AudioFileSourceID3.h розкриває клас, спеціально розроблений для спрощення розбору метаданих ID3 у аудіофайлах або потоках, керованих фреймворком Arduino. Він діє як обгортка навколо іншого об'єкта AudioFileSource, прозоро обробляючи вилучення ID3-тегів, делегуючи основну функціональність читання аудіоданих. Такий підхід спрощує розробку додатків, надаючи уніфікований інтерфейс для доступу як до метаданих, так і до аудіоконтенту.

Коли викликається функція читання AudioFileSourceID3, вона перехоплює потік даних з базового об'єкта AudioFileSource. Під час початкового читання вона ретельно перевіряє наявність заголовка ID3, який зазвичай ідентифікується підписом «ID3». Якщо цей підпис виявлено, клас вступає в дію, розбираючи заголовок для вилучення такої важливої інформації, як версія ID3, загальний розмір у потоці та наявність байтів несинхронізації (у деяких версіях використовується для запобігання неправильним інтерпретаціям під час синтаксичного розбору). Цей початковий синтаксичний аналіз забезпечує клас знаннями, необхідними для точної інтерпретації наступного потоку даних.

Після розбору заголовка AudioFileSourceID3 входить у ретельний цикл, перебираючи потенційні кадри ID3. Кожен кадр діє як контейнер, інкапсулюючи певний тип метаданих, таких як назва альбому, ім'я виконавця або номер треку. Клас ретельно зчитує ідентифікатор кадру (зазвичай 4 байти), щоб визначити тип

інформації, яку він містить. Потім він отримує розмір кадру, визначаючи кількість байт, відведених для цього конкретного запису метаданих. Клас також перевіряє стиснення кадру, хоча наданий код наразі не працює зі стисненими кадрами. Нарешті, він витягує фактичний вміст кадру на основі розміру кадру і прапорця несинхронізації, якщо це можливо. На цьому етапі може знадобитися прочитати додаткові байти, щоб врахувати потенційні неузгодженості, спричинені несинхронізацією.

Після того, як вміст кадру отримано, AudioFileSourceID3 використовує свої знання про ідентифікатори кадрів для перетворення необроблених байтів на інформацію, придатну для читання людиною. Наприклад, якщо ідентифікатор кадру відповідає назві альбому, клас витягує відповідні символи і, ймовірно, використовує функцію зворотного виклику (cb.md у коді), щоб повідомити програму про знайдені метадані. Ця функція зворотного виклику, розміщена в окремій частині кодової бази, відповідає за обробку витягнутих метаданих, наприклад, за відображення їх у користувацькому інтерфейсі або збереження для подальшого використання.

Після ретельної обробки всіх зустрінутих ID3-кадрів клас досягає кінця сегмента даних ID3 у потоці. Щоб повідомити програмі, що наступний потік даних складається з власне аудіовмісту, AudioFileSourceID3, ймовірно, використовує ту саму функцію зворотного виклику (cb.md), яка сигналізує про кінець ID3-тегів. Це сповіщення, по суті, діє як передача даних, повідомляючи програмі, що тепер вона може зосередитися на отриманні та обробці необроблених аудіобайтів.

Прозоре читання аудіоданих. Після початкової перевірки ID3 (або якщо заголовок ID3 не знайдено), функція читання плавно переходить до своєї основної функціональності: делегування фактичного читання аудіоданих базовому об'єкту AudioFileSource. Це делегування забезпечує безперервний потік даних до програми. Додаток отримує уніфікований потік, на початку якого плавно інтегровані метадані ID3 (через функцію зворотного виклику) (якщо вони є), а далі йдуть необроблені аудіобайти, що представляють власне аудіоконтент.

3.2 Висновок

Розділ, присвячений реалізації програмних і технічних засобів для керування звуковими кодеками на базі мікроконтролера ESP8266, надає вичерпний і детальний посібник з процесу розробки. У цьому розділі висвітлено практичні кроки, зроблені для створення та управління різними джерелами аудіофайлів за допомогою реалізації спеціальних заголовних файлів та пов'язаних з ними C++-файлів. Нижче наведено основні висновки з цього розділу. У розділі систематично розглянуто розробку та реалізацію численних компонентів, необхідних для роботи з аудіофайлами. Сюди входять вихідні файли для різних аудіоформатів і потоків даних, таких як FATFS, HTTP-потоки, ICY-потоки та метадані ID3.

Розділ містить ретельні деталі про кожен файл і функцію, що гарантує, що читач може слідувати за кодом і розуміти призначення і функцію кожної частини. Така ретельність має вирішальне значення для відтворення налаштування або його модифікації для різних застосувань. Завдяки підтримці різних джерел звуку система стає універсальною та надійною, здатною керувати різними типами аудіовходів. Така гнучкість корисна для розробки більш комплексних додатків для управління звуком. Використання мікроконтролера ESP8266, відомого своєю економічністю та бездротовими можливостями, робить проект доступним і практичним для багатьох користувачів. Можливості мікроконтролера добре використовуються для ефективного вирішення завдань обробки звуку. Таким чином, цей розділ надає основу для розуміння та реалізації систем керування аудіо з використанням мікроконтролера ESP8266. Вона пропонує детальний і структурований підхід до розробки програмного забезпечення, гарантуючи, що читач зможе ефективно керувати і контролювати різні аудіокодеки за допомогою добре організованої і модульної системи кодування.

ВИСНОВКИ

Перший розділ присвячено важливим моментам, що пояснюють значення та потенційні переваги розробки програмного забезпечення системи підтримки мультимедіа для мікроконтролерних систем на базі ATMega328.

Другий розділ присвячено всебічному огляду компонентів, програмного забезпечення та технічних засобів, необхідних для керування звуковими кодеками за допомогою мікроконтролера ESP8266. Розділ починається зі знайомства з бібліотекою ESP8266Audio, висвітлення її ролі та значення в управлінні аудіоопераціями на платформі ESP8266.

Третій розділ присвячений реалізації програмних і технічних засобів для управління звуковими кодеками на базі мікроконтролера ESP8266 і містить вичерпний і детальний посібник з процесу розробки. У цьому розділі висвітлено практичні кроки, зроблені для створення та управління різними джерелами аудіофайлів за допомогою реалізації спеціальних заголовних файлів і пов'язаних з ними C++-файлів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Fu S., Bhavsar P. C. Robotic Arm Control Based on Internet of Things. *IEEE Long Island Systems Applications and Technology Conf.* 2019. Vol. 1. Pp. 1-6.

2. Rao B. N., Sudheer R. Energy Monitoring using IOT. Int. Conf. on Inventive Computation Technologies. 2020. Pp. 868-872.

3. Mesquita J., Guimarães D., Pereira C., Santos F., Almeida L. Assessing the ESP8266 WiFi module for the Internet of Things. *IEEE Int. Conf. on Emerging Technologies and Factory Automation*. 2018. Pp. 784-791.

4. Lekhaa T. R., Rajeshwari S., Sequeira J. A., Akshayaa S. Intelligent Shopping Cart Using Bolt Esp8266 Based on Internet of Things. *Int. Conf. on Advanced Computing Communication Systems*. 2019. Pp. 758-761.

5. de A. Lima C. M., da Silva E. A., Velloso P. B. Performance Evaluation of 802.11 IoT Devices for Data Collection in the Forest with Drones. *IEEE Global Communications Conf*. 2018. Pp. 1-7.

6. Maier A., Sharp Y., Vagapov Y. Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things. *Internet Technologies and Applications*. 2017.

7. Kodali R. K., Soratkal S. MQTT based home automation system using ESP8266. *IEEE Region 10 Humanitarian Technology Conf*. 2016. Pp. 1-5.

8. Yadav R. K., Vohra H. Design architecture and comparison of interactive smart button using HC-05 and ESP8266. *Int. Conf. on Computing Communication and Automation*. 2017. Pp. 982-985.

9. Ertam F., Kilincer I. F., Yaman O., Sengur A. A New IoT Application for Dynamic WiFi based Wireless Sensor Network. *Int. Conf. on Electrical Engineering*. 2020. Pp. 1-4.

10. Mosashvili, Oniani S. Distance Monitoring System with ESP8266 for Industrial Automation Machines. *IEEE Int. Symposium on Smart and Wireless Systems within the Conf. on Intelligent Data Acquisition and Advanced Computing Systems.* 2020. Pp. 1-5.

11. Starodubtsev, Gajniyarov I., Samedov R., Sibogatova A., Antipina I., Zolotareva Y. Animatronic Hand Model on the Basis of ESP8266. *Int. Multi-Conf on Engineering Computer and Information Sciences*. 2019. Pp. 500-503.

12. Khanchuea, Siripokarpirom R. A Multi-Protocol IoT Gateway and WiFi/BLE Sensor Nodes for Smart Home and Building Automation: Design and Implementation. *Int. Conf. of Information and Communication Technology for Embedded Systems*. 2019. Pp. 1-6.

13. Kodali R. K., Mahesh K. S. Low cost ambient monitoring using ESP8266. *Int. Conf. on Contemporary Computing and Informatics*. 2016. Pp. 779-782.

14. Rizal Isnanto R., Eko Windarto Y., Imago Dei Gloriawan J., Noerdiyan Cesara F. Design of a Robot to Control Agricultural Soil Conditions using ESP-NOW Protocol. *Int. Conf. on Informatics and Computing*. 2020. Pp. 1-6.

15. Pertab Rai, Rehman M. ESP32 Based Smart Surveillance System. *Int. Conf. on Computing Mathematics and Engineering Technologies*. 2019.

16. Minchev D., Dimitrov A. Home automation system based on ESP8266. *Int. Symposium on Electrical Apparatus and Technologies*. 2018.

17. Mahmood, Exel R., Bigler T. On clock synchronization over wireless LAN using timing advertisement mechanism and TSF timers. *IEEE Int. Symposium on Precision Clock Synchronization for Measurement Control and Communication*. 2014. Pp. 42-46.

18. Mahmood, Exel R., Sauter T. A simulation-based comparison of IEEE 802.11s timing advertisement and SyncTSF for clock synchronization. *IEEE WorId Conf. on Factory Communication Systems*. 2015. Pp. 1-4.

20. Hoang T. N., Van S., Nguyen B. D. ESP-NOW Based Decentralized Low Cost Voice Communication Systems For Buildings. *Int. Symposium on Electrical and Electronics Engineering*. 2019. Pp. 108-112.

21. ESP-NOW/ URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html (Дата звернення: 5.04.20240)..

22. ESP-NOW - ESP-IDF Programming Guide v4.0-dev-1191. 2019. Pp. 2-4.

23. Kumar S., et al. Internet of Things is a revolutionary approach for future technology enhancement: a review. Journal of Big Data. 2019. Vol. 6. Pp. 1–21.

24. Noura M., Atiquzzaman M., Gaedke M. Interoperability in Internet of Things: *Taxonomies and Open Challenges. Mobile Network Application*. 2019. Vol. 24, No. 3. Pp. 796–809.

25. IoT application areas. https://iot-analytics.com/top-10-iot-project-application-areas-q3-2016/ Accessed 8 Apr 2020.

26. Wadhwani S., Singh U., Singh P., Dwivedi S. Smart Home Automation and Security System using Arduino and IOT. *Int. Research Journal of Engineering and Technology (IRJET)*. 2018. Vol. 5, No. 2. Pp. 1357–1359.

27. Pasha S. Thingspeak Based Sensing and Monitoring System for IoT with Matlab Analysis. *Int. Journal of New Technology and Research (IJNTR)*. 2016. Vol. 2, No. 6. Pp. 19-23.

28. Kiliç T., Bayir E. An Investigation on Internet of Things Technology (IoT) In Smart Houses. *Int. Journal of Engineering Research and Development*. 2017. Vol. 9. Pp. 197–206.

29. Çeltek S. A., Soy H. An application of building automation system based on wireless sensor/actuator networks. *Int. Conf. on Application of Information and Communication Technologies (AICT)*. 2015. Pp. 450-453.

30. Wahab N. A., et al. Performance of Environmental and Energy Audit for Manufacturing Industrial Buildings. *Indonesian Journal of Electrical Eng. and Computer Science (IJEECS)*. 2018. Vol. 12, No. 2. Pp. 534-541.

31. Chaudhury S., Paul D., Mukherjee R., Haldar S. Internet of Thing based healthcare monitoring system. 2017.

32. 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON). Bangkok. 2017. Pp. 346-349.

33. Visconti P., et al. IoT-oriented software platform applied to sensors-based farming facility with smartphone farmer app. *Bulletin of Electrical Engineering and Informatics (BEEI)*. 2020. Vol. 9, No. 3. Pp. 1095-1105.

34. Salih N. A. J., Hasan I. J., Abdulkhaleq N. I. Design and implementation of a smart monitoring system for water quality of fish farms. *Indonesian Journal of Electrical Engineering and Computer Science (IJEECS)*. 2019. Vol. 14, No. 1. Pp. 44-50.

35. Van De Moosdijk J., Visser D. Car security: remote keyless entry and go. Revision 232. June 2009.

36. Mason S. Vehicle remote keyless entry systems and engine immobilisers: Do not believe the insurer that this technology is perfect. *Computer Law & Security Review*. 2012. Vol. 28. Pp. 195-200.

37. Agarwal R., Boominathan P. Vehicle Security System Using IoT Application. *Int. Research Journal of Engineering and Technology (IRJET)*. 2018. Vol. 5, No. 4. Pp. 910-912.

38. Datasheet Espressif Systems. Espressif Smart Connectivity Platform: ESP8266. 2013. URL: https://www.electroschematics.com/wp-content/uploads/2015/02/esp8266-datasheet.pdf (Дата звернення: 5.04.20240)..

39. Datasheet ESP 12E PIN OUT. URL: https://components101.com/wireless/esp12e-pinout-datasheet; https://www.make-it.ca/nodemcu-arduino/nodemcu-details-specifications/.

40. Arduino software. URL: https://www.arduino.cc/.

41. Arduino IDE online version. URL: https://www.arduino.cc/en/Main/Software (Дата звернення: 5.04.20240)..

42. Arduino firebase Master. URL: https://github.com/FirebaseExtended/firebase-arduino (Дата звернення: 5.04.20240)..

43. ArduninoJson. URL: https://github.com/bblanchon/ArduinoJson/tree/5.x (Дата звернення: 5.04.20240)..

44. DHT by adafruit. URL: https://github.com/adafruit/DHT-sensor-library.

45. Adafruit universal sensor. URL: https://github.com/adafruit/Adafruit_Sensor (Дата звернення: 5.04.20240)..

46. ESP8266 wi-fi module. URL: https://github.com/esp8266/Arduino (Дата звернення: 5.04.20240)..

47. Alsalemi A., et al. Real-Time Communication Network Using Firebase Cloud IoT Platform for ECMO Simulation. *IEEE Int. Conf. on Internet of Things (iThings), Green Computing and Communications (GreenCom), Cyber, Physical and Social Computing (CPSCom), Smart Data (SmartData)*. 2017. Pp. 178-182.

48. This system- IoT and Cloud based system Database Data Retreived as Json Document. URL: https://nubarealtimedb.firebaseio.com/.json.

49. Firebase Rest API Usage. URL: https://firebase.google.com/docs/reference/rest/database#section-cond-ifmatch.

50. MIT Tool. URL: http://appinventor.mit.edu/explore/.

51. MIT project space for the app development of this IoT and Cloud based system. URL: http://ai2.appinventor.mit.edu/#5948786283777395 2 (Дата звернення: 5.04.20240).

Додаток А

Блок-схеми алгоритмів

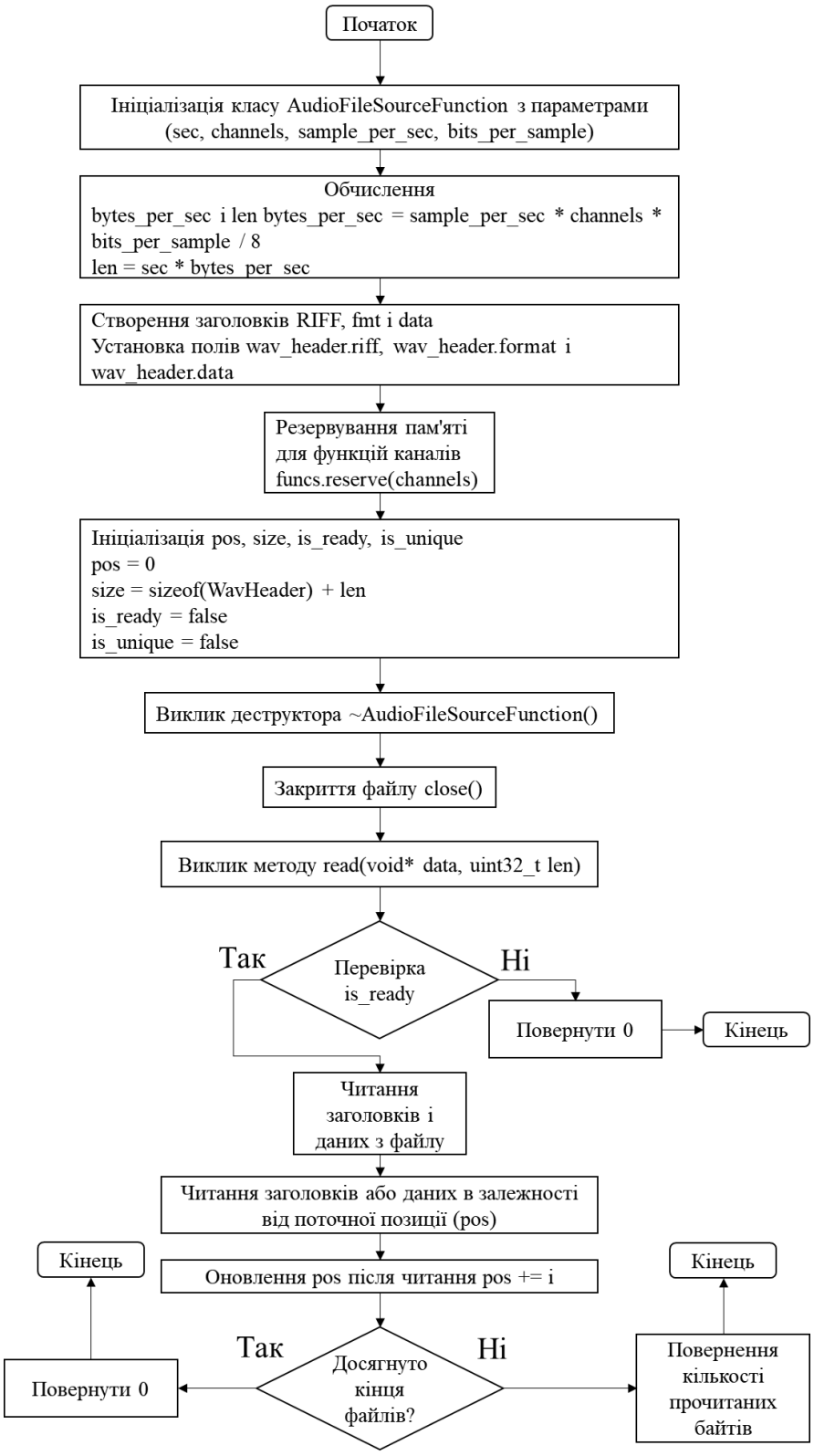


Рисунок А.1 - AudioFileSourceFunction.cpp

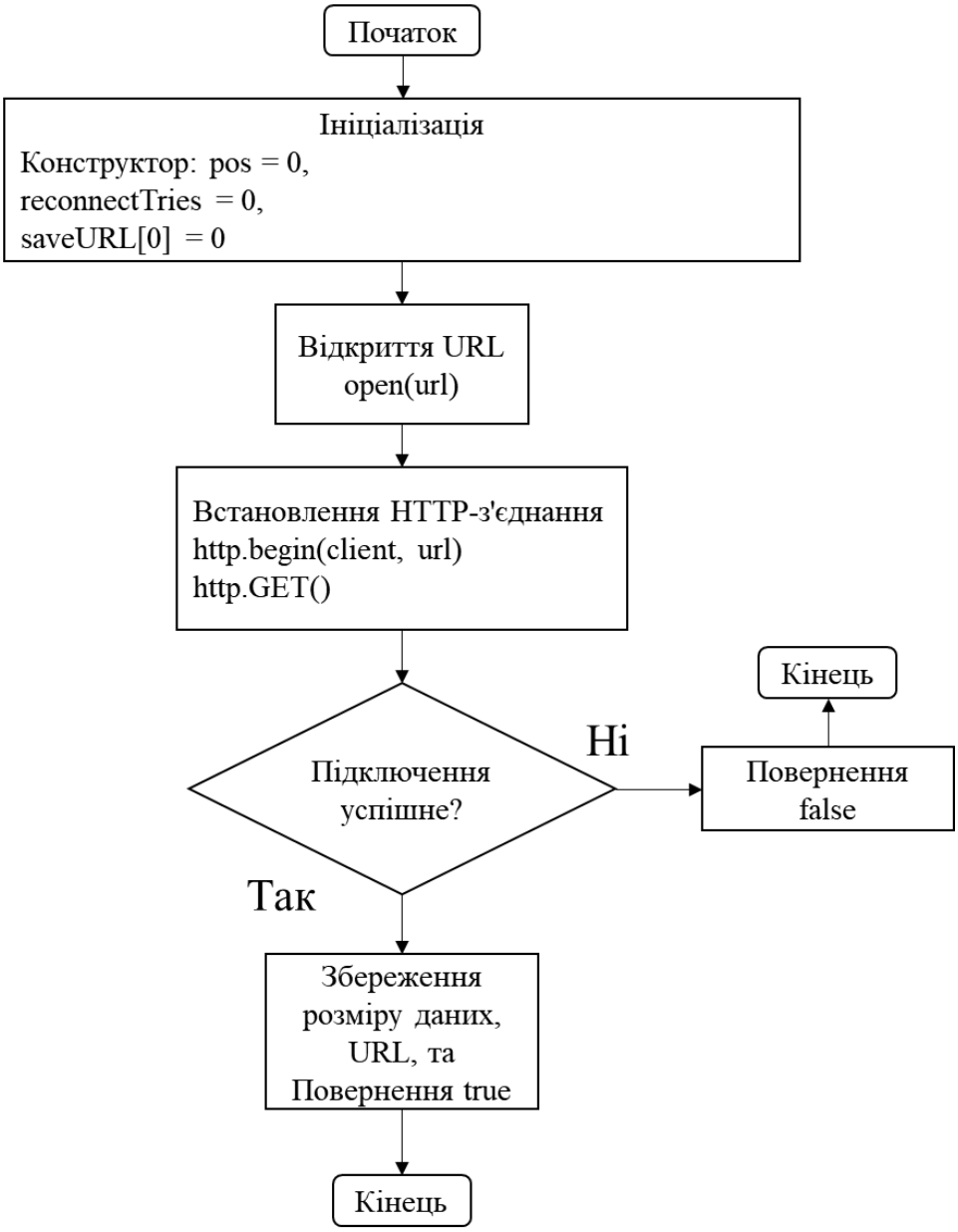


Рисунок А.2 - AudioFileSourceHTTPStream.cpp

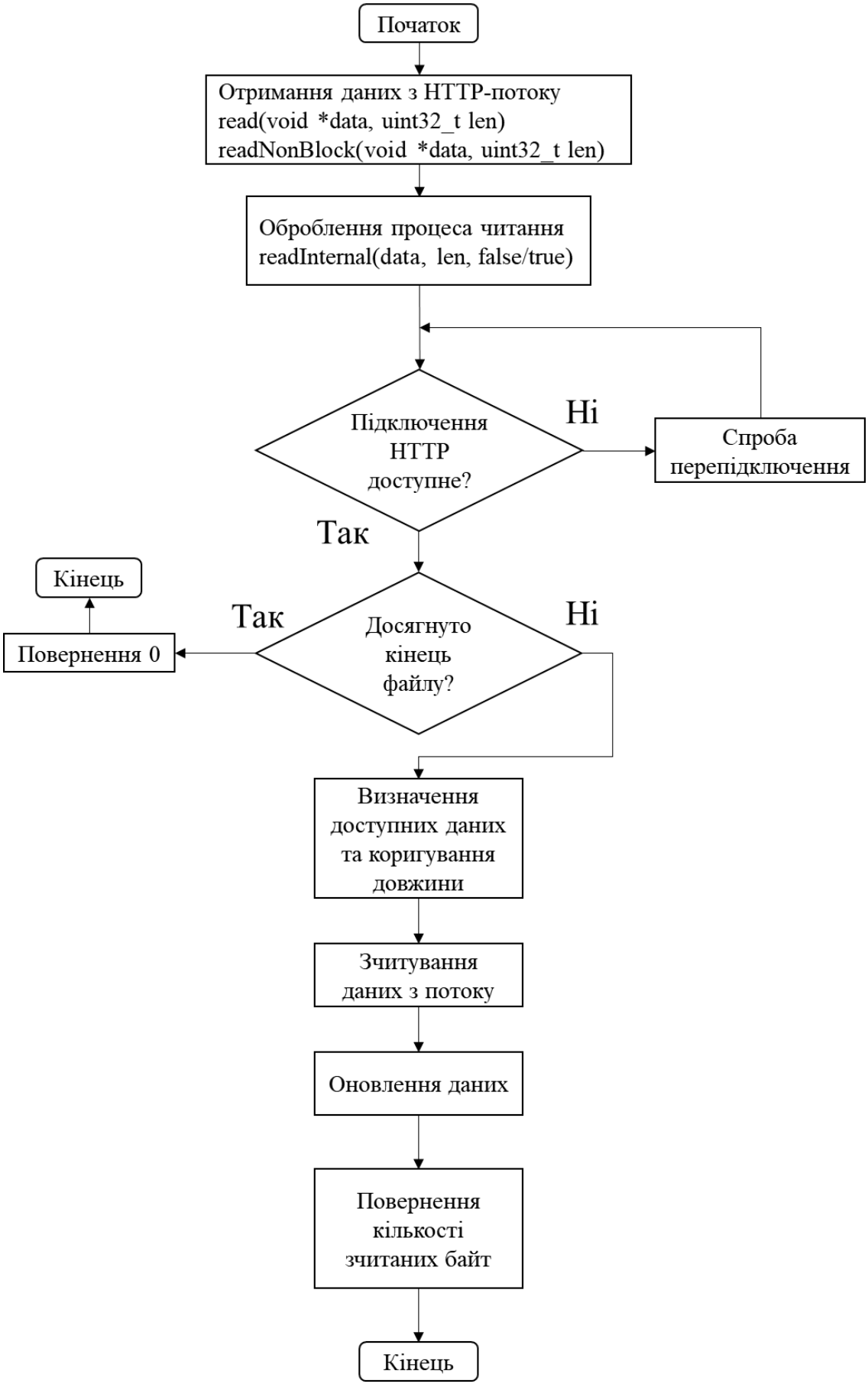


Рисунок А.3 - AudioFileSourceHTTPStream.cpp

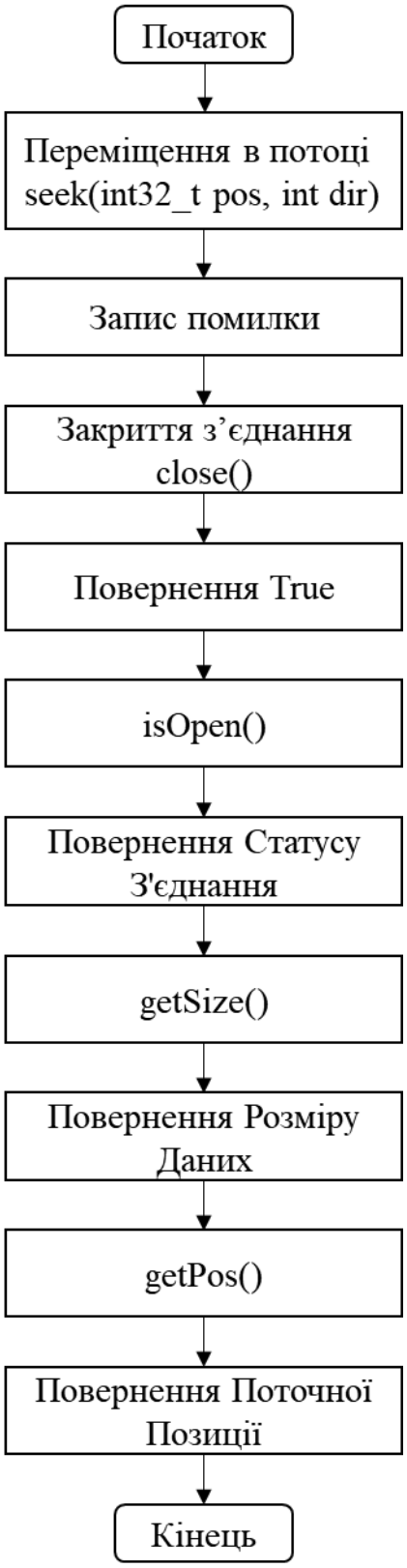
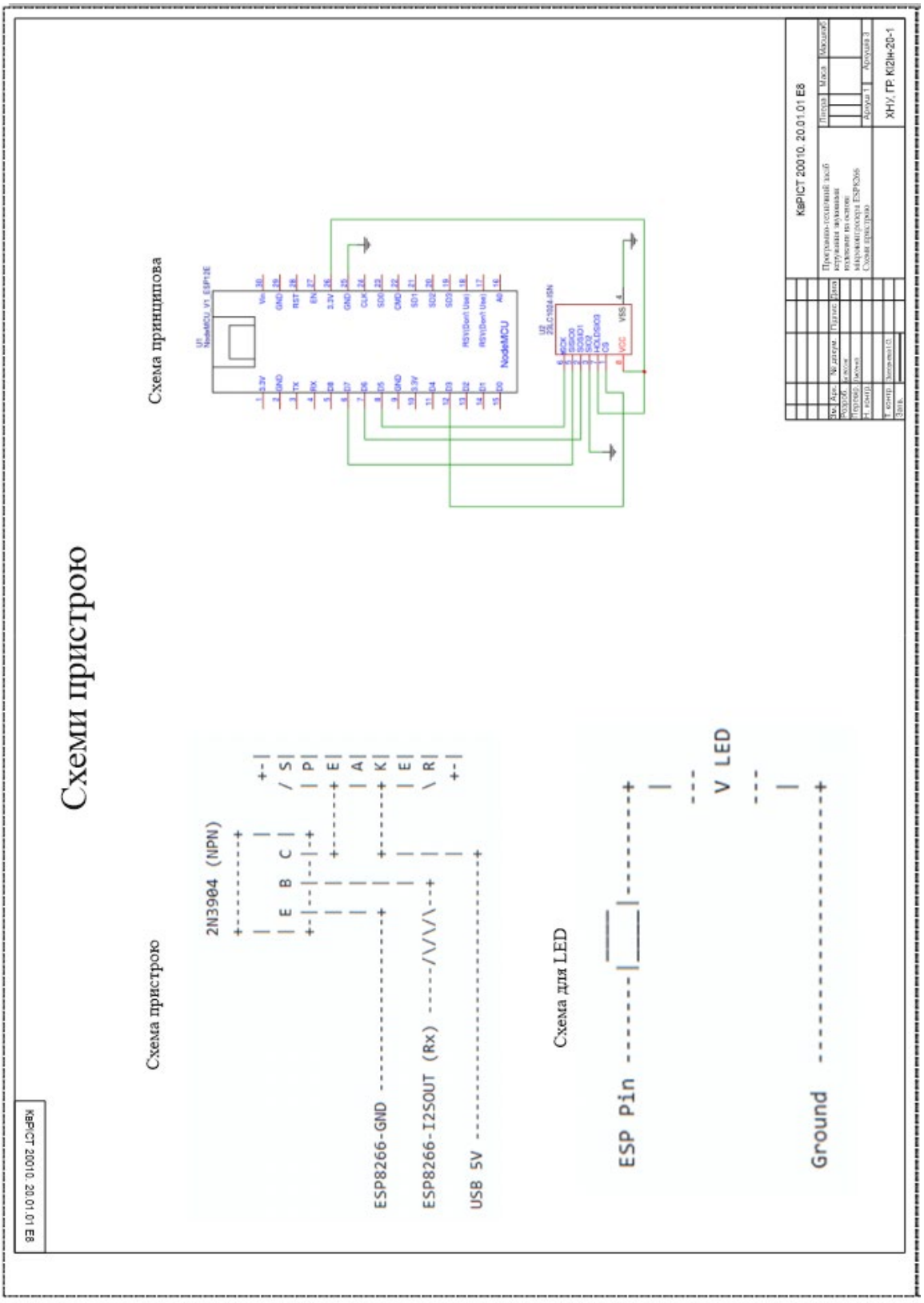


Рисунок А.4 - AudioFileSourceHTTPStream.cpp

Додаток Б

(обов’язковий)

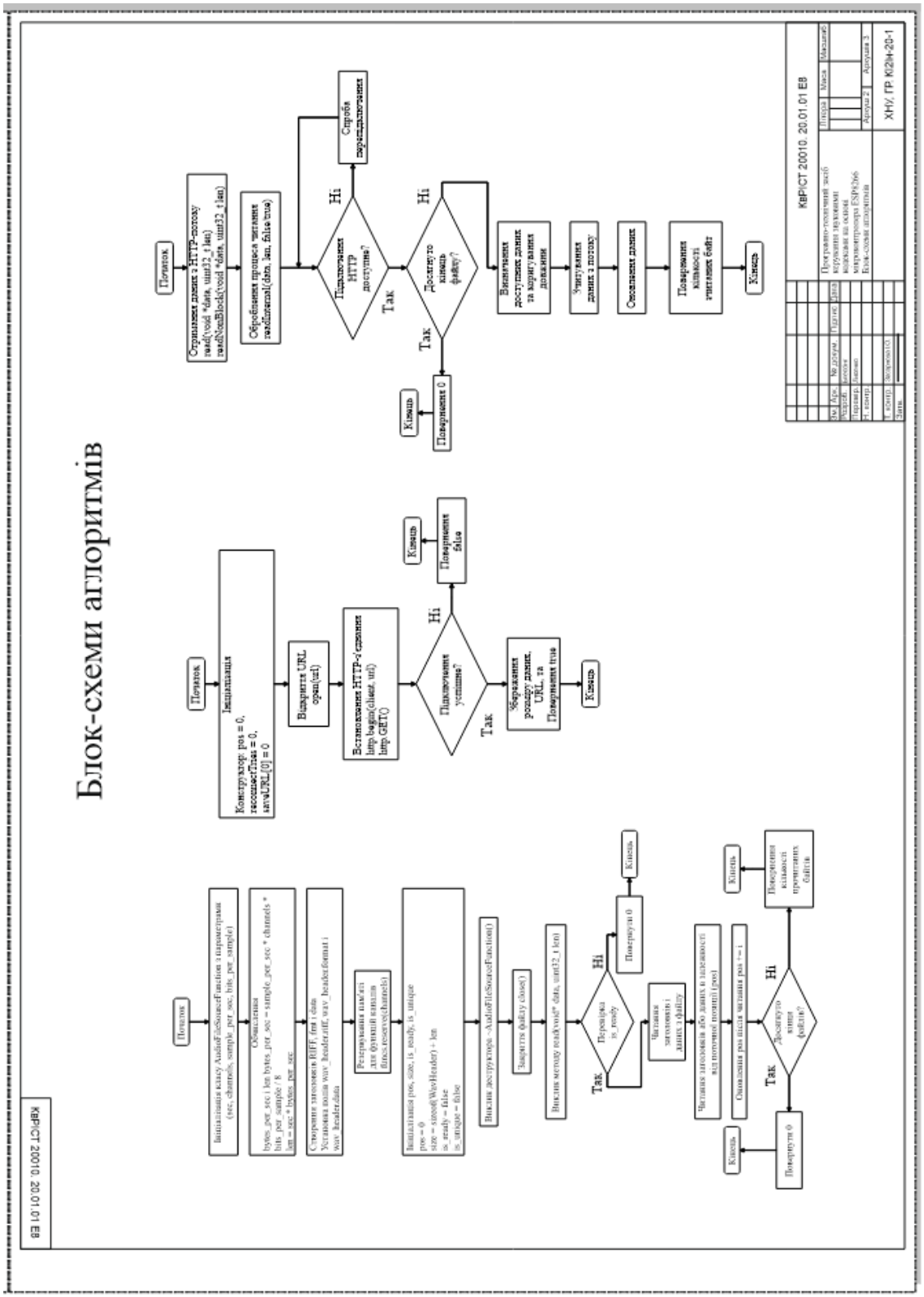
Копія креслення «Схеми пристрою»



Додаток В

(обов’язковий)

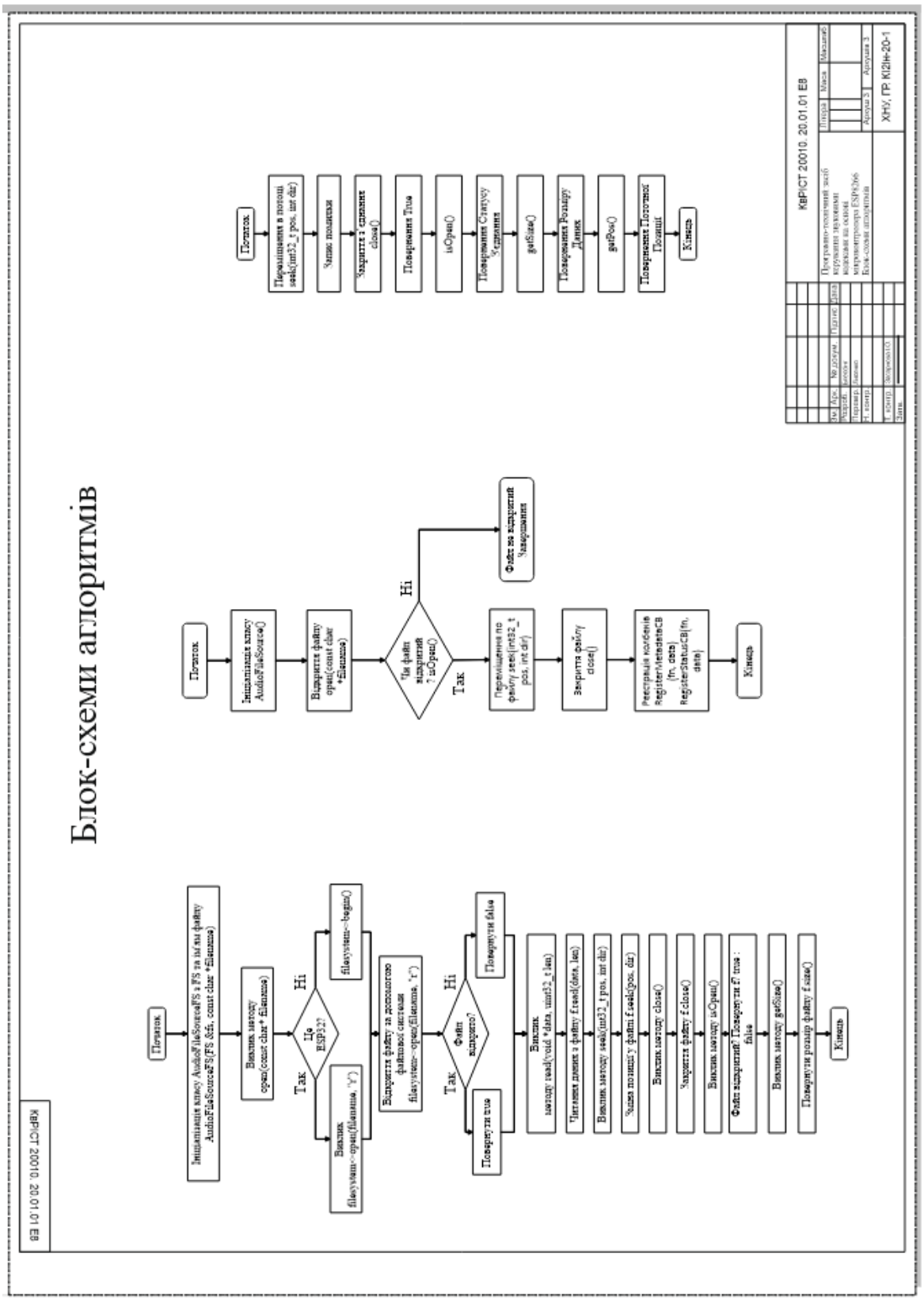
Копія креслення «Блок-схеми алгоритмів»



Додаток Г

(обов’язковий)

Копія креслення «Блок-схеми алгоритмів»



Khmelnytsky National University
Faculty of Information Technology
Department of Computer Engineering and Information Systems

QUALIFICATION WORK
bachelor
Educational level

Software and technical tool for controlling sound codecs based on the ESP8266 microcontroller
topic

QWCE 20010. 20.01.01 NE
code

Branch of knowledge *12 "Information technologies"*
Code, name

Specialty *123 "Computer Engineering"*
Cipher, name

Educational program *"Computer engineering and programming"*
Name

Performed by: IV year student, group КІ2ін-20-1 _____ Kahoka Blessing
Signature Initials, surname

Head _____ S.M. Lysenko
Signature, date Initials, surname

Normocontroller _____ I.O. Zasornova
Signature, date Initials, surname

I admit to protection:
Chief computer department
engineering and information
systems _____ T.O. Govorushchenko
Signature Initials, surname

"30" May 2024

Khmelnytskyi 2024

Khmelnytskyi National University

Faculty OF INFORMATION TECHNOLOGIES

Department OF COMPUTER ENGINEERING AND INFORMATION SYSTEMS

BACHELOR'S level of education

Field of knowledge 12 INFORMATION TECHNOLOGIES

Specialty 123 COMPUTER ENGINEERING

Educational program "COMPUTER ENGINEERING AND PROGRAMMING"

I APPROVE

Chief departments

T.O. Govorushchenko

" 10 " 01 2024

TASK
ON BACHELOR'S QUALIFICATION WORK

Kahotsi Blessings

Surname, first name, patronymic of the student

1. Topic of the project (work) Software and technical tool for controlling sound codecs based he the ESP8266 microcontroller

Project manager (works) S.M. Lysenko, doctor of technical sciences, prof.

Surname, first name, patronymic, academic degree, academic

Approved by the order of the rector of the university dated 15 02.2024 No. 8

2. The deadline for student submission of a project (work) to the department is June 1, 2024.

3. Initial data for the project (work) Assignment for qualification work

4. Content of the explanatory note (list of issues to be developed)

Multimedia support systems based he sound codecs

Description of the components for the software and technical tool for controlling sound codecs based on the esp8266 microcontroller

Implementation of the software and technical tool for controlling sound codecs based he the esp8266 microcontroller

5. List of graphic material (indicating mandatory drawings)

Device schemes

The flowchart of the algorithm

The flowchart of the algorithm

6. Consultants of sections of the diploma project (work)

Section	Surname, initials and position of the consultant	Signature, date	
		issued the task	accepted the task
Standard control	Zasornova I.O., Ph.D, associate professor		
Anti-plagiarism	Nicheporuk A.O., Ph.D, associate professor		

7. Issue date of the assignment "_10_" _____01_____2024

CALENDAR PLAN

No. z/p	Name of stages (sections) diploma project (work)	Deadline stages of the project (work)	Note
1	Choosing a research direction and agreeing the topic of the qualification work with the supervisor	10.01.2024	done
2	Acquaintance with the subject area; formulation of the goal and objectives of the research; definition of the object and subject of research	01.02.2024	done
3	Work on chapter 1 - research of the subject area and formulation of the problem	01.03.2024	done
4	Work on section 2 - selection of components for designing a system of adaptive application of monitoring elements of a reconnaissance UAV	04.01.2024	done
5	Work on chapter 3 – designing a system of adaptive application of monitoring elements of a reconnaissance UAV	04.29.2024	done
6	Issuance of an explanatory note according to the requirements	05.25.2024	done
7	Preliminary protection of the WKR	05.26.2024	done
8	Protection of the WKR at the meeting of the EC	June 2024	

Student _____ K. Blessing
Signature Initials, surname

Head of work _____ S.M. Lysenko
Signature Initials, surname

№ of Row	format	Designation	Name	Am. of sheets	№ instance	Note
			Text documents			
1		QWCE. 20010. 20.01.01 EN	Explanatory note	62		
			Graphic materials			
2		QWCE. 20010. 20.01.01 E8	Device schemes	1		
3		QWCE. 20010. 20.01.01 E8	The flowchart of the algorithm	1		
4		QWCE. 20010. 20.01.01 E8	The flowchart of the algorithm	1		

M.	Let	№ of doc	Sign.	Date	QWCE.20010. 20.01.01 EN		
Author		Blessing			Project information	Letter: E	Sheet: 1 / Sheets: 1
Supervisor		Lysenko				KhNU, КІін-20-1	
Reg. contr.		Zasornova					
Approve		Hovorushchenko		12.06			

ANNOTATION

The subject of the qualification work: "Software and technical tool for controlling sound codecs based on the ESP8266 microcontroller".

The author of the work: Kahoka Blessing.

Head of work: Lysenko Sergii.

Explanatory note: 62 pp., 12 figures, 1 table, 4 appendices, 51 sources.

Graphic part: 3 drawings.

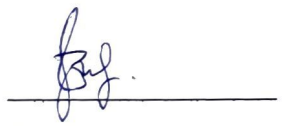
SOUND CODEC MANAGEMENT SOFTWARE, MICROCONTROLLER, ESP8266.

The aim of the thesis is to develop a software and technical tool for controlling sound codecs based on the ESP8266 microcontroller.

The object of the research is the control of sound codecs based on the ESP8266 microcontroller.

The subject of the study is a software and technical tool for controlling sound codecs based on the ESP8266 microcontroller.

During the conduct of this study, the method of systematic literature review was used to study and analyze the subject area of this study from text sources of information.



Student signature

10.06.2024

Date

CONTENT

INTRODUCTION 5

1 MULTIMEDIA SUPPORT SYSTEMS BASED ON SOUND CODECS 6

1.1 The concept of sound codecs 6

1.2 Relevance of the use of sound codecs 6

1.2.1 Codingand decoding 6

1.2.2 Compression data 7

1.2.3Applicationlossy and lossless compression (Lossy and Lossless Compression) 7

1.2.4 Relevanceuse of sound codecs 7

1.3 The process of data compression in audio codecs 8

1.4 Types of codecs 9

1.5 Known software system libraries for encoding and decoding audio codecs 10

1.6 Use of audio codecs in Internet of Things (IoT) systems 12

1.7 Relevance of using microcontrollers for Internet of Things (IoT) systems 13

1.8 Control of sound codecs based on the ESP8266 microcontroller 14

1.9 Conclusion 16

2 DESCRIPION OF THE COMPONENTS FOR THE SOFTWARE AND TECHNICAL TOOL FOR CONTROLLING SOUND CODECS BASED ON THE ESP8266 MICROCONTROLLER 18

2.1 SP8266Audio 18

2.2 Installation 19

2.3 AudioInput 19

2.4 AudioFileSource classes 20

2.5 AudioGenerator classes 22

2.6 AudioOutput classes 23

2.7 Adafruit I2S DAC and PCM5102 DAC 24

2.9 Software I2S Delta-Sigma DAC (i.e. playing music with a single transistor and speaker) 25
2.10 High pitched buzzing with the 1-T circuit 27
2.11 Debugging the 1-T amp circuit 27
2.12 SPDIF optical output 28
2.13 Using external SPI RAM to increase buffer 29
2.14 Usage of the SD cards and ESP8266Audio 29
2.15 Porting to other microcontrollers 29
2.16 Conclusion 30
3 IMPLEMENTATION OF THE SOFTWARE AND TECHNICAL TOOL FOR CONTROLLING SOUND CODECS BASED ON THE ESP8266 MICROCONTROLLER 33
3.1 Software implementation 33
3.1.1 AudioFileSource header file implementation 33
3.1.2 AudioFileSourceBuffer header file implementation 37
3.1.3 AudioFileSourceBuffer header file implementation 39
3.1.4 AudioFileSourceFATFS header file implementation 40
3.1.5 AudioFileSourceHTTPStream header file implementation 42
3.1.6 AudioFileSourceFS header file implementation 45
3.1.7 AudioFileSourceFunction.cpp 46
3.1.8 AudioFileSourceFunction header file implementation 49
3.1.9 AudioFileSourceHTTPStream.cpp 51
3.1.10 AudioFileSourceHTTPStream header implementation 53
3.1.11 AudioFileSourceICYStream.cpp 56
3.1.12 AudioFileSourceICYStream.h 58
3.1.13 AudioFileSourceID3.cpp 60
3.1.14 AudioFileSourceID3 62
3.2 Conclusion 64
CONCLUSIONS 66
REFERENCES 67
ANNEX A THE FLOWCHART OF THE ALGORITHMS 72

ANNEX B COPY OF THE DRAWING “DEVICE SCHEME” 76
ANNEX C COPY OF THE DRAWING “FLOWCHART OF THE ALGORITHMS”..... 77
ANNEX D COPY OF THE DRAWING «FLOWCHART OF THE ALGORITHMS»..... 78

INTRODUCTION

The relevance of system software (SW) to support multimedia in microcontroller systems is growing in the modern world every day, and here is why:

1. Increasing functionality of microcontrollers. Modern microcontrollers are becoming more and more powerful and functional. They are no longer limited to simple tasks, but are also used in many applications, including multimedia, such as audio, video, and graphics processing.

2. Expanding applications of microcontrollers. Microcontroller systems are used in a wide range of applications, including the automotive industry, medical devices, home appliances, industrial systems, and more. In many of these areas, it is important to be able to process and display multimedia content.

3. Increasing requirements for visualization and interactivity. In many cases, users and systems require high quality sound, images, and other multimedia elements to provide better interaction and visualization of information.

4. Development of the Internet of Things (IoT). In the IoT network, a large number of devices require processing and transmission of multimedia data. For example, video cameras, audio systems, smart home devices - all of them require multimedia support for efficient operation.

5. Virtualization and use of graphical interfaces. GUI systems require multimedia support for effective virtualization and display of graphics, which allows for more convenient and intuitive devices.

6. Expanded capabilities of sound systems. In many applications, sound quality is important, such as in automotive systems, home audio systems, medical devices, and more. Multimedia support is key to achieving high sound quality.

All of these factors make it important to develop and implement system software for microcontrollers that provides multimedia support, including video codecs, graphics libraries, and other components for optimal processing and playback of multimedia content.

1 MULTIMEDIA SUPPORT SYSTEMS BASED ON SOUND CODECS

1.1 The concept of sound codecs

A sound codec (from "codec" - coder-decoder) is software or hardware that encodes and decodes audio signals. The main purpose of an audio codec is to reduce the amount of data without losing significant audio quality.

It is important to understand that audio signals in digital form can take up much more space than analog signals. Audio codecs are used to compress audio data for more efficient storage and transmission. They can be classified into two main categories:

Lossy codecs (Lossy Codecs). These codecs compress audio data by removing some information that is considered less important to the human ear. Examples of lossy codecs are MP3, AAC, and Ogg Vorbis.

Lossless Codecs. They compress audio data without losing quality. They reduce the amount of data, but at the same time you can restore the original audio signal without any loss. Examples of lossless codecs are FLAC, ALAC, and WAV.

Audio codecs are used in various fields such as audio recording, audio transmission over networks, audio streaming, and in various multimedia devices.

1.2 Relevance of the use of sound codecs

An audio codec is software or hardware capable of encoding or decoding audio signals. This is typically used to compress audio data to reduce its size when transmitted or stored.

1.2.1 Codingand decoding

An audio codec is capable of converting audio signals into a digital format during encoding and restoring them to their original form during decoding. This process is important to ensure efficient storage or transfer of audio data.

1.2.2 Compression data

One of the main functions of an audio codec is to compress audio data to reduce the size of a file or data stream. This is especially important in today's networks and devices where data volume is a critical factor.

1.2.3Applicationlossy and lossless compression (Lossy and Lossless Compression)

Some codecs lose some information during compression (lossy compression), which results in some loss of audio quality. Other codecs retain all information (lossless compression), but may take up more space.

Understanding different compression algorithms such as MPEG, AAC, MP3, FLAC, etc. is important. Each of these algorithms has its advantages and disadvantages, which affect the quality and volume of compressed audio data.

1.2.4 Relevanceuse of sound codecs

Topicalitythe use of audio codecs is in the following areas:

1. Network transmission.
2. Media storage.
3. Audio processing.
4. Mobile devices.
5. Audio and video conferences.

In today's world, where streaming services, video conferencing and online games are the norm, audio codecs play a key role in the transmission of audio data over the Internet. Effective compression allows you to maintain high-quality sound while minimizing bandwidth.

Audio codecs are also used to compress and store audio files on devices and in cloud services.

In the field of audio processing and music production, sound codecs allow you to work with audio data, reducing its volume without significant loss of quality.

Audio codecs are used in smartphones, headphones and other mobile devices to optimize space for storing and transmitting audio data.

Codecs are used to provide quality sound in video conferencing platforms where it is important to have a clear and realistic sound.

Understanding audio codecs is an important element, especially if they are aimed at software or hardware developmentfor processing and transmission of audio data.

1.3 The process of data compression in audio codecs

Data compression in audio codecs is the process of reducing the volume of audio information using various methods, which allows you to save an audio file or audio signal with a smaller number of bytes.

This allows you to reduce bandwidth for transmitting audio data over networks, save storage space, and ensure efficient use of resources.

Audio codecs use two main types of data compression: lossy and lossless.

During lossy compression, some audio information is lost to reduce the amount of data.

This method is widely used in audio codecs such as MP3, AAC, WMA, and others. They use different algorithms, such as removing unnecessary fragments, psychoacoustic models, and others, to compress audio without significant loss of quality for most listeners.

Advantages of the method: effective amount of compression, well suited for streaming and storing music on devices with limited memory.

Disadvantages of the method: loss of quality, especially with a high degree of compression. But for most people the losses are imperceptible or acceptable.

During lossless compression, all audio data is preserved exactly and can be restored in its original form.

Lossless compression is used in audio codecs such as FLAC, ALAC, and APE. This is useful for situations where high quality without data loss is important, such as in a music studio or archival storage.

Advantages of the method: Fully reproduces the original audio information without loss of quality.

Disadvantages of the method: Lossless files usually take up more space than their lossy counterparts, making them less practical for storage on devices with limited memory or for transmission over a network with limited bandwidth.

The choice between lossy and lossless compression depends on specific needs, which may include audio quality, storage space availability,and data transfer rate.

1.4 Types of codecs

There are many types of audio codecs, each of which has its own characteristics and applications. Below are several types of audio codecs that are popular and widely used.

MP3 (MPEG Audio Layer III). Type: lossy. Application: One of the most common formats for audio compression. It is used to store music and audio content in various devices.

AAC (Advanced Audio Coding). Type: lossy.

Often used in media players, smartphones, streaming platforms such as Apple Music and Spotify.

FLAC (Free Lossless Audio Codec). Type: lossless. Provides fully lossless compression, often used in audiophile audio systems and for storing archival copies of audio files.

ALAC (Apple Lossless Audio Codec). Type: lossless. Application: Developed by Apple, used in environments where compatibility with Apple products is important.

WAV (Waveform Audio File Format). Type: uncompressed (lossless) or compressed (lossy).

It is often used to store high-quality uncompressed audio recordings. Can also be used with various compression codecs.

Opus. Type: lossy or lossless. Optimized for real-time use such as VoIP, streaming video and audio chat.

Vorbis. Type: lossy. A free codec used in video games and Internet radio.

Dolby Digital (AC-3). Type: lossy. Widely used for encoding audio in video files, DVDs, and television.

DTS (Digital Theater Systems). Type: lossy. Used in movie theaters, home theaters and various media players.

Monkey's Audio (APE). Type: Lossless. Used to compress audio files without loss of quality.

Choosing a specific audio codec depends on specific needs, such as quality, file size, device support, and other factors.

1.5 Known software system libraries for encoding and decoding audio codecs

A program library is a set of functions and procedures that can be used by programs to perform certain tasks without having to write code from scratch. The library includes ready-to-use code that the developer can call in his program to perform specific operations.

Basic concepts related to software libraries:

1. Functions. The library contains functions that can be called from program code. Functions perform certain tasks and allow the developer to use ready-made code for certain operations.

2. Methods. The term "method" is used when talking about object-oriented libraries. A method is a function that is associated with a specific object or class in a program.

3. Classes and objects. Some libraries may include classes and objects that provide a framework for organizing code and enable an object-oriented approach to programming.

4. Interfaces. Interfaces define the way a program interacts with a library. If a library has a well-defined interface, it makes it easier to use and integrate into other programs.

5. Documentation. A library is usually accompanied by documentation that explains how to use various functions and methods. Documentation is important for developers using the library.

6. Versions. Libraries may have different versions that include bug fixes, new features, and improvements. Developers should specify which version of the library they are using.

Software libraries are widely used to facilitate the software development process and promote code reuse. They can be written in different programming languages and provide a convenient interface to use their functions and methods.

Different libraries are available for encoding and decoding audio codecs, depending on the programming language and platform. Below are some of the most common libraries for parsing and decoding audio codecs:

1. FFmpeg. Language: C. FFmpeg is a powerful multimedia data processing library that supports audio and video decoding, encoding, transcoding, and streaming. It supports a wide range of audio codecs, making it a popular choice for multimedia projects.

2. Libsndfile. Language: C. Libsndfile is a C library designed to work with various audio file formats. It supports reading and writing audio files and provides a simple interface for decoding and encoding audio data.

3. Opus. Language: C. Opus is an open source, free audio codec designed for efficient low-latency audio compression and streaming. It is suitable for a wide range of applications and is commonly used for voice over IP (VoIP) and real-time communication.

4. Vorbis. Language: C. Vorbis is an open source program: Vorbis is a free and open source audio codec developed by the Xiph.Org Foundation. It provides high-quality audio compression and is commonly used for streaming and internet radio.

5. MPG123. Language: C. MPG123 is a fast and freely distributed audio codec: MPG123 is a fast, free, open source MPEG audio decoder library. It is designed for decoding audio files and streams in MPEG format and is widely used in various audio programs.

6. AudioToolbox. Language: Objective-C (iOS/macOS). AudioToolbox is an Apple development framework for iOS and macOS. It includes an API for decoding and encoding audio files, supporting various formats such as AAC, MP3 and others.

7. JavaZoom JLayer. Language: Java. JLayer is a Java library for decoding MP3 files. It provides a simple API for working with MP3 audio data and is commonly used in Java applications that require MP3 decoding functionality.

8. GStreamer. Language: C (with bindings for different languages). GStreamer is a multimedia framework that supports the construction of graphs for processing multimedia components. It includes plug-ins for various codecs that allow you to decode and encode audio and video.

Factors such as the programming language you're using, the specific audio codecs you need to support, and the licensing requirements for your project should be considered when choosing a library.

Also, check for compatibility with the target platform and the level of community support for ongoing maintenance and updates.

1.6 Use of audio codecs in Internet of Things (IoT) systems

The use of audio codecs in Internet of Things (IoT) systems can be diverse and include the following aspects:

1. Sound Alerts and Alarms. Audio codecs can be used to play beeps or notifications. This can be useful for informing users about various events or device status.

2. Voice Commands and Responses. Embedded IoT systems can use audio codecs to process and recognize user voice commands or to provide voice responses.

3. Multimedia Applications. In some cases, especially in leisure or entertainment devices, audio codecs may be used to play music or other multimedia content.

4. Telephony and Video communication. Some IoT devices may have telephony or video capabilities, and audio codecs are used to process and transmit audio data.

5. Analysis of Sound Signals. In some cases, IoT devices may use audio codecs for audio signal analysis, such as noise detection, voice recognition, or audio ambient monitoring.

6. Volume Detectors and Sound Sensors. In some applications, audio codecs are used as sensitive sensors to measure loudness or to detect specific sound patterns.

7. Monitoring and Security Systems. Applications of audio codecs may include monitoring and security systems where sound may be used to detect emergencies or events.

8. Language Assessment Systems. Audio codecs can be used to create speech evaluation systems that analyze the quality and characteristics of pronunciation.

It is important to consider that the choice of audio codecs in IoT systems is related to resource constraints, such as power, memory and computing power, which may affect the choice of a particular solution.

1.7 Relevance of using microcontrollers for Internet of Things (IoT) systems

Microcontrollers remain key components for Internet of Things (IoT) systems and are relevant for several reasons:

1. Low Cost and Small Size. Microcontrollers are characterized by low manufacturing costs and small dimensions, which make them ideal for embedded systems with limited resources, which is characteristic of many IoT devices.

2. Low energy consumption. Many microcontrollers are optimized for low power consumption, which is an important aspect for wireless IoT devices that can run on batteries.
3. Built-in Peripheral Devices. Microcontrollers often have built-in peripherals such as analog-to-digital converters (ADCs), timers, and communication interfaces, which facilitates their integration into IoT systems.
4. Wireless support. Many microcontrollers support various wireless technologies such as Wi-Fi, Bluetooth, Zigbee, LoRa, which makes them ideal for connecting to IoT networks.
5. Ease of programming. Many microcontrollers have well-developed development tools and programming environments that allow developers to easily create programs to use them.
6. Availability of a wide range. There are a large number of microcontrollers available from different manufacturers, allowing developers to choose the device that best suits their specific needs and project requirements.
7. Simplifying software development. Microcontrollers are commonly used in simple embedded systems, which simplifies software development for IoT devices compared to more powerful microprocessors or microcomputers.
8. Protection and Security Capabilities. Some microcontrollers include attack protection and have built-in tools to keep IoT devices secure.
9. Wide range of applications. Microcontrollers are used in a variety of industries, such as automotive, medical, industrial, home appliances, and others, which proves their versatility.

In summary, microcontrollers remain relevant for IoT systems due to their advantages in resource optimization, energy efficiency, and wireless support.

1.8 Control of sound codecs based on the ESP8266 microcontroller

To implement audio codec control based on the ESP8266 microcontroller, you first need to determine the specific audio codec you plan to use, as different codecs may require different approaches.

However, a general approach to implementing audio codec management might include the following steps:

1. Selecting the audio codec. Choose the sound codec that suits your requirements and the capabilities of the ESP8266. The most common codecs include PCM, I2S, or analog audio output capabilities.

2. Connecting the sound codec to the ESP8266. Ensure that the audio codec is properly connected to the ESP8266 microcontroller using the necessary interfaces (eg I2S or SPI for digital codecs, or analog pins for analog codecs).

3. Connection and sound configuration. Use the SDK or framework for ESP8266 to set up connectivity and sound configuration. This may include setting the correct settings (resolution, sample rate, etc.) for the audio output.

4. Management software development. Write software for the ESP8266 that allows you to control the audio output. This may include code for initialization, reading and playing audio files, responding to control commands, analyzing audio signals, etc.

5. Sound processing. If necessary, enable sound processing functions such as equalizer, noise reduction, or other effects depending on your usage.

6. Testing and debugging. Test the system, make sure that the sound is output correctly and meets the requirements. Adjust the settings as necessary.

7. Integration into an IoT project. Integrate the developed software into your IoT project. Ensure interaction with other modules and functionality of the device.

8. Ensuring security and optimization. Consider security issues, in particular, if the device is connected to the Internet. Optimize your code to make efficient use of ESP8266 resources.

9. Documentation and support. Document the software you develop and provide support for other developers who can work with your code.

This process may vary depending on the specific codec, framework, or SDK you're using, but the general approach remains the same for many ESP8266 audio solutions.

1.9 Conclusion

In the chapter several important points were carefully considered in order to clarify the significance and potential benefits of developing multimedia support system software for ATMega328-based microcontroller systems.

The development of system software (SSP) to support multimedia on ATMega328-based microcontroller systems can have its applications and relevance in a number of scenarios. However, one should take into account the limited resources of these microcontrollers, in particular in the area of memory and computing capabilities. Below we will consider some options and scenarios where the development of such an SDR may be appropriate:

The development of SDRs for processing audio data can be relevant for creating audio projects such as simple synthesizers, recording audio signals, sound effects, and other audio-like applications.

Using a microcontroller to develop simple musical instruments or sequencers where audio signal processing is important.

Providing support for sound visualization through LEDs (e.g. response to music) can be relevant for entertainment and art installations.

In some cases, where it is necessary to process or display video data on the display, it is possible to consider the development of simple systems, for example, to display animations or text information.

Development of simple interactive games or game projects where sound or graphic elements are important.

It is worth noting that in modern conditions, more powerful platforms such as ARM Cortex-M or the use of specialized chips that have more resources for such tasks are often used for complex tasks in the field of multimedia on microcontrollers.

Thus, the relevance of developing system software for multimedia on ATMega328 microcontrollers depends on the specific task and requirements of the project, as well as on the readiness to work with the limited resources of this microcontroller.

2 DESCRIPION OF THE COMPONENTS FOR THE SOFTWARE AND TECHNICAL TOOL FOR CONTROLLING SOUND CODECS BASED ON THE ESP8266 MICROCONTROLLER

2.1 SP8266Audio

An Arduino library designed to parse and decode audio files such as MOD, WAV, MP3, FLAC, MIDI, AAC, and RTTL, facilitating playback on an I2S DAC or a software-simulated delta-sigma DAC with dynamic 32x-128x oversampling.

The library offers full support for the ESP8266, with extensive maturity, and also provides substantial support for the ESP32, utilizing both its built-in DAC and external DACs.

For autonomous, real-time speech synthesis, consider using ESP8266SAM. This library, which builds on the aforementioned library, leverages an old formant-based synthesis program to enable speech capabilities on the ESP8266, requiring minimal memory and no network connection.

The MOD routines were sourced from StellarPlayer, and the MP3 routines from libMAD.

The AAC decoding code comes from the Helix project, licensed under RealNetwork's RSPL. For commercial use, you will still need AAC licensing from Via Licensing.

On the ESP32, AAC-SBR is supported, which many web radio stations use to further reduce bandwidth. Unfortunately, the ESP8266 does not support AAC-SBR due to insufficient onboard RAM.

MIDI decoding is achieved through a heavily adapted MIDITONES combined with a significantly memory-optimized TinySoundFont. Refer to the respective source files for more details.

Opus, OGG, and OpusFile libraries are from Xiph.org, licensed under the Xiph license, with patent details available in src/{opusfile,libggg,libopus}/COPYING.

Opus decoding currently only works on the ESP32 due to its high memory requirements.

The ESP8266Audio library was used for project.

To get started, we hato ensure we are using version 2.6.3 or later of the Arduino libraries for ESP8266, or the latest ESP32 SDK from Espressif.

2.2 Installation

Install the library in to the directory

```
~/Arduino/libraries
mkdir -p ~/Arduino/libraries
cd ~/Arduino/libraries
git clone https://github.com/myP8266Audio
When in the IDE please select the following options on the ESP8266:
Tools->lwIP Variant->v1.4 Open Source, or V2 Higher Bandwidth
Tools->CPU Frequency->160MHz
```

2.3 AudioInput

Create an AudioInputXXX source that points to your input file and an AudioOutputXXX sink, which can be configured as either an I2S, I2S-sw-DAC, or "SerialWAV" to write a WAV file to the Serial port for later use on your development system.

Additionally, set up an AudioGeneratorXXX to handle decoding the input and sending it to the output.

After setting these up, you need to call the AudioGeneratorXXX::loop() function within your main loop() one or more times.

This function reads the necessary parts of the file, fills the I2S buffers, and returns immediately.

Because this process is not interrupt-driven, having long delay() calls in your code can cause playback hiccups.

To avoid this, either break large delays into smaller segments with calls to AudioGenerator::loop() in between or reduce the sampling rate to lower the number of samples needed per second.

Refer to the examples directory for basic demonstrations.

The snippet below illustrates how to play an MP3 file using the simulated I2S DAC:

2.4 AudioFileSource classes

AudioFileSource: This serves as a foundational class that implements a straightforward read-only "file" interface. Its necessity arises from the varied filesystem implementations across Arduino, each with its own unique attributes.

By utilizing this wrapper, the complexities of individual filesystems can be abstracted, thereby simplifying the AudioGenerator, which exclusively relies on these straightforward functions.

AudioFileSourceSPIFFS is designed to read files from the SPIFFS filesystem in the controller.

AudioFileSourcePROGMEM is designed to read files from a PROGMEM array. On UNIX systems, one can utilize "xxd -i file.mp3 > file.h" to obtain the basic format. Following this, adding "const" and "PROGMEM" to the generated array and including it in the sketch is necessary.

AudioFileSourceHTTPStream is the implementation facilitates simple streaming HTTP reading for ShoutCast-type MP3 streaming.

While it is functional, it is not yet fully resilient. Moreover, at 44.1kHz 128-bit, it may stutter due to CPU limitations, albeit it generally operates as intended.

AudioFileSourceBuffer - Double buffering, useful for HTTP streams.

AudioFileSourceBuffer serves as an input source that supplements the output of any other AudioFileSource with an additional RAM buffer.

This feature is especially beneficial for web streaming scenarios, where maintaining 1-2 packets in memory is crucial to ensure smooth playback without interruptions.

To implement this, first, create your standard input file source.

Then, create the buffer with the original source as its input. Finally, pass this buffer object to the generator.

This setup ensures that the audio data is buffered appropriately, enhancing the streaming experience and minimizing playback disruptions.

```
AudioGeneratorMP3 *mp3;
AudioFileSourceHTTPStream *file;
AudioFileSourceBuffer *buff;
AudioOutputI2SNoDAC *out;
...
  // Create the HTTP stream normally
  file = new AudioFileSourceHTTPStream("http://your.url.here/mp3");
  // Create a buffer using that stream
  buff = new AudioFileSourceBuffer(file, 2048);
  out = new AudioOutputI2SNoDAC();
  mp3 = new AudioGeneratorMP3();
  // Pass in the *buffer*, not the *http stream* to enable buffering
  mp3->begin(buff, out);
...
```

AudioFileSourceID3 - ID3 stream parser filter with a user-specified callback.

This class accepts any other AudioFileSource as input and produces an AudioFileSource that is compatible with any decoder.

Additionally, it automatically extracts ID3 tags from MP3 files. You must define a callback function, which will be invoked as tags are decoded.

This enables you to update your user interface (UI) state with this information dynamically. For further details, refer to the PlayMP3FromSPIFFS example.

2.5 AudioGenerator classes

AudioGenerator serves as the foundational class for all file decoders. It requires an AudioFileSource and an AudioOutput object to retrieve data from and write decoded samples to, respectively.

To ensure uninterrupted playback without skipping, call its loop() function as frequently as possible to keep the buffers adequately filled.

AudioGeneratorWAV is the decoder that reads and plays Microsoft WAVE (.WAV) format files of 8 or 16 bits.

AudioGeneratorMOD is designed to read and play Amiga ModTracker files (.MOD). To handle the numerous SPIFFS reads required for raw instrument sample data retrieval, it's recommended to use a 160MHz clock.

AudioGeneratorMP3 is a specifically for reading and playing MP3 format files (.MP3) using a ported libMAD library.

To decode 128KBit 44.1KHz MP3s smoothly, a 160MHz clock is advisable.

AudioGeneratorFLACis a decoder plays FLAC files via a ported libflac-1.3.2 library. It typically requires around 30KB of heap and minimal stack.

AudioGeneratorMIDI utilizes a wavetable synthesizer and a SoundFont2 wavetable input to play MIDI files.

While theoretically capable of handling up to 16 simultaneous notes, the actual number may vary depending on the memory required for the SF2 structures.

AudioGeneratorAAC requires approximately 30KB of heap and plays mono or stereo AAC files using the Helix fixed-point AAC decoder.

AudioGeneratorRTTTL offers the enjoyment of monophonic, 4-octave ringtones on your ESP8266 with very low memory and CPU requirements for simple tunes.

2.6 AudioOutput classes

AudioOutput serves as the foundational class for all output drivers.

It processes one sample at a time and returns true or false based on whether there is buffer space available for it.

If false is returned, it is the responsibility of the calling object (AudioGenerator) to retain the data that didn't fit and attempt again later.

AudioOutputI2S is an interface and is designed for any I2S 16-bit DAC.

It sends stereo or mono signals at the set frequency. Tested with Adafruit's I2SDAC and a Beyond9032 DAC from eBay, functioning reliably up to 44.1KHz.

To utilize the internal DAC on ESP32, instantiate this class as AudioOutputI2S(0,1).

AudioOutputI2SNoDAC is a class that exploits the I2S interface to play music without a DAC, effectively converting it into a 32x or higher oversampling delta-sigma DAC.

Use the provided schematic to drive a speaker or headphone from the I2STx pin (Rx).

Depending on the transistor used, you may need to disconnect the Rx pin from the driver for serial uploads. This interface supports mono-only output.

AudioOutputSPDIF (experimental): Another interface leveraging the I2S peripheral to transmit BMC encoded S/PDIF bitstream. To interface with an S/PDIF receiver, it requires an optical or coaxial transceiver.

It should function even with a basic setup with a red LED and current-limiting resistor fed into a TOSLINK cable.

The minimum supported sample rate is 32KHz. Due to BMC coding, the actual symbol rate on the pin is 4x the normal I2S data rate, which quickly depletes DMA buffers.

AudioOutputSerialWAV: Writes binary WAV format data with headers to the Serial port. If you capture the serial output to a file, you can play it back on your development system.

AudioOutputSPIFFSWAV is a class that writes binary WAV format data with headers to a SPIFFS filesystem. Ensure the filesystem is mounted and SPIFFS is started before calling. Use the SetFilename() function to select the output file before starting.

AudioOutputNull discards samples to /dev/null. Primarily used for speed testing as it doesn't artificially limit the AudioGenerator output speed since there are no buffers to fill/drain.I2S DACs. Project uses the Adafruit I2S +3W amp DAC and a generic PCM5102 based DAC.

The essential pins are presented in the table 2.1.

Table 2.1 – The essential pins

I2S pin	Common label*	ESP8266 pin
LRC	D4	GPIO2
BCLK	D8	GPIO15
DIN	RX	GPIO3

The "common label" column provided here is applicable to widely used NodeMCU and D1 Mini development boards. However, it's worth noting that some manufacturers may use different mappings, so the labels listed here might not correspond exactly to your specific model.

2.7 Adafruit I2S DAC and PCM5102 DAC

This setup is straightforward and requires only the GND, VIN, LRC, BCLK, and DIN pins to be connected. We have to take sure to use +5V on the VIN pin to achieve the

maximum volume. Various versions of PCM5102 DAC boards and regardless of the form factor, they've all had the same pinout were used. Beyond the I2S interface itself, there are several input configuration pins that need to be wired:

- Connect the 3.3V from ESP8266 to VCC, 33V, and XMT.
- Connect the GND from ESP8266 to GND, FLT, DMP, FMT, and SCL.
- For the standard I2S interface:
- Connect BCLK to BCK.
- Connect I2SO to DIN.
- Connect LRCLK (WS) to LCK.

There are numerous variants available, and most should work adequately with this code and the ESP8266. Additionally, remember to tie off any unused inputs to either GND or VCC as necessary. Leaving an input pin floating on any integrated circuit can lead to unstable operation as it may pick up environmental noise due to its very low input capacitance, potentially causing disruption with internal IC settings.

2.9 Software I2S Delta-Sigma DAC (i.e. playing music with a single transistor and speaker)

For optimal audio quality, especially in stereo, investing in a genuine I2S DAC is recommended. Adafruit offers an excellent mono option with an amplifier, while stereo unamplified DACs can be found inexpensively on eBay or other platforms.

However, if you're looking for a budget-friendly solution, the software delta-sigma DAC with 32x oversampling (up to 128x if the audio rate permits) still provides satisfactory sound quality. To implement this, AudioOutputI2SNoDAC object instead of AudioOutputI2S in your code was used. Additionally, refer to the following schematic to drive a 2-3W speaker using a single inexpensive NPN 2N3904 transistor and a ~1K resistor 2N3904 (NPN).

Device scheme is presented in Figure 2.1.

Another consideration for improved performance is to include a 220uF capacitor from USB5V to GND. This addition helps filter out any voltage droop that may occur during high-volume playback, contributing to smoother operation.

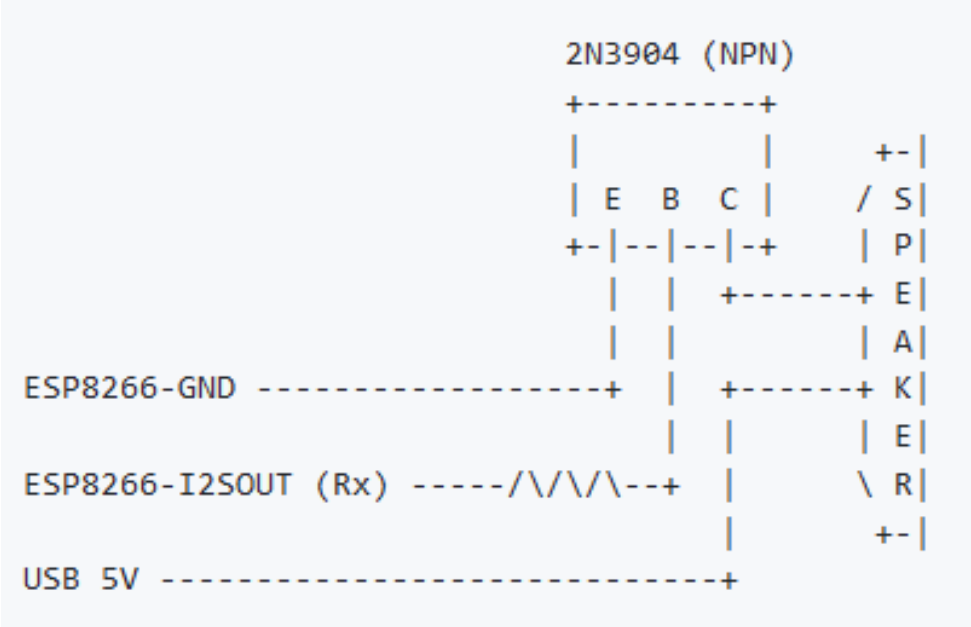


Figure 2.1 – Device scheme

If a 5V source isn't available on your ESP model, you can utilize the 5V from your USB serial adapter, or alternatively, the 3V from the ESP8266 (although this may result in lower volume output).

It's crucial not to attempt to drive the speaker directly without the transistor. The pins on the ESP8266 cannot supply sufficient current to drive even a headphone effectively, and attempting to do so may risk damaging your device.

Here are the connections as presented in figure 2.2.

```
ESP8266-RX(I2S tx) -- Resistor (~1K ohm, not critical) -- 2N3904 Base
ESP8266-GND        -- 2N3904 Emitter
USB-5V             -- Speaker + Terminal
2N3904-Collector   -- Speaker - Terminal
```

Figure 2.2 - Connections

In a previous version of this schematic, there was a direct connection from the ESP8266 to the base of the transistor.

While this setup does provide maximum amplitude, it can also draw more current from the ESP8266 than is safe, potentially causing the transistor to overheat.

In the latest ESP8266Audio release, with the software delta-sigma DAC, the LRCLK and BCLK pins are now available for application use. You can use normal pinMode, digitalWrite, or digitalRead functions as desired.

2.10 High pitched buzzing with the 1-T circuit

It's important to note that the 1-T amp cannot drive any type of amplified speaker. If the speaker requires power, USB input, has lights, Bluetooth, or a battery, it cannot be used with this circuit.

The 1T output produces a binary signal at either 0V or 5V, without any intermediary levels. When connected directly to an 8-ohm paper physical speaker, the inertia of the speaker cone acts as a low-pass filter, averaging the density of pulses to produce a smooth, analog output. However, when the 1T output is connected to an amplifier, it alternates between grounding and overdriving the op-amp's input at a high frequency. This can cause ringing, and since op-amps typically have a high-frequency response, they amplify the high-frequency noise, resulting in buzzing.

The same issue may occur with piezo speakers, which have a very high-frequency response and little to no inertia. This can lead to buzzing at high frequencies.

One workaround is to connect the 1T output to a low-pass filter and then feed that into an amplifier.

However, at this point, it may be simpler to just use an I2S DAC, which avoids these issues altogether and provides stereo and true 16-bit output.

2.11 Debugging the 1-T amp circuit

If you've assembled the amplifier but aren't getting any sound, it has outlined a helpful debugging sequence to troubleshoot the issue.

Double-check the wiring to ensure correctness. GPIO pins and board pins may not always align and can vary significantly between brands of ESP8266 carrier boards.

We have to verify that the transistor is connected properly. Refer to the datasheet for the specific package style and ensure that the leads are connected correctly. Note that in this package style, the lead that is solitary in the middle of one side is the collector, not the base as one might expect.

Confirm if there is approximately 5 volts between the collector and emitter of the transistor. Consider whether the transistor may have been damaged or overheated during soldering or due to improper connection. Perform an out-of-circuit diode check voltage drop test using a multimeter from base to emitter and base to collector. The voltage drop should be between 0.5 and 0.7 volts. If it's shorted, open, or conducting in both directions, replace the transistor and ensure it is connected correctly.

2.12 SPDIF optical output

The recommended method is to use an optical TOSLINK transmitter, such as TOTXxxx. For testing purposes, you can try using a ~660nm red LED and a resistor.

This setup is similar to your basic Blink project with an external LED, except that the LED will blink at a slightly faster rate (Figure 2.3).

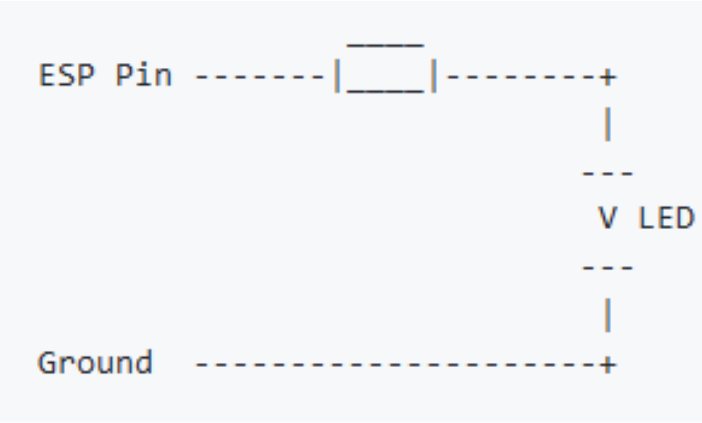


Figure 2.3 – Scheme for LED

For ESP8266 with a red LED (with a ~1.9V forward voltage drop), you'll need a minimum 150-ohm resistor (to limit the current to a maximum of 12mA per pin).

The output pin is fixed and should be connected to GPIO3/RX0.

On the ESP32, the output pin can be configured using AudioOutputSPDIF(gpio_num), allowing for greater flexibility in pin selection.

2.13 Using external SPI RAM to increase buffer

A class has been implemented to facilitate the use of a 23LC1024 SPI RAM from Microchip as an input buffer. This chip connects to the ESP8266 HSPI port and provides a sizable buffer to mitigate playback hiccups when streaming web content.

The current version allows for the utilization of the standard hardware CS (GPIO15) or any other pin through software, albeit with slightly reduced performance. Below is an example schematic illustrating this setup (Figure 2.4).

2.14 Usage of the SD cards and ESP8266Audio

Wemos SD card shield utilizes GPIO15 as the SD chip select. This presents an issue because GPIO15 is also assigned to I2SBCLK and is driven even if you're utilizing the NoDAC option. To resolve this, we need to change the chip select (CS) to another pin and update your program accordingly. Once this adjustment is made, your setup should function correctly.

2.15 Porting to other microcontrollers

The AudioGenerator routines do not contain ESP8266-specific code, making porting to other controllers relatively straightforward, provided they have the same

endianness as the Xtensa core used. If we're considering porting to another controller, feel free to reach out to me. I may be able to offer guidance to help steer you in the right direction.

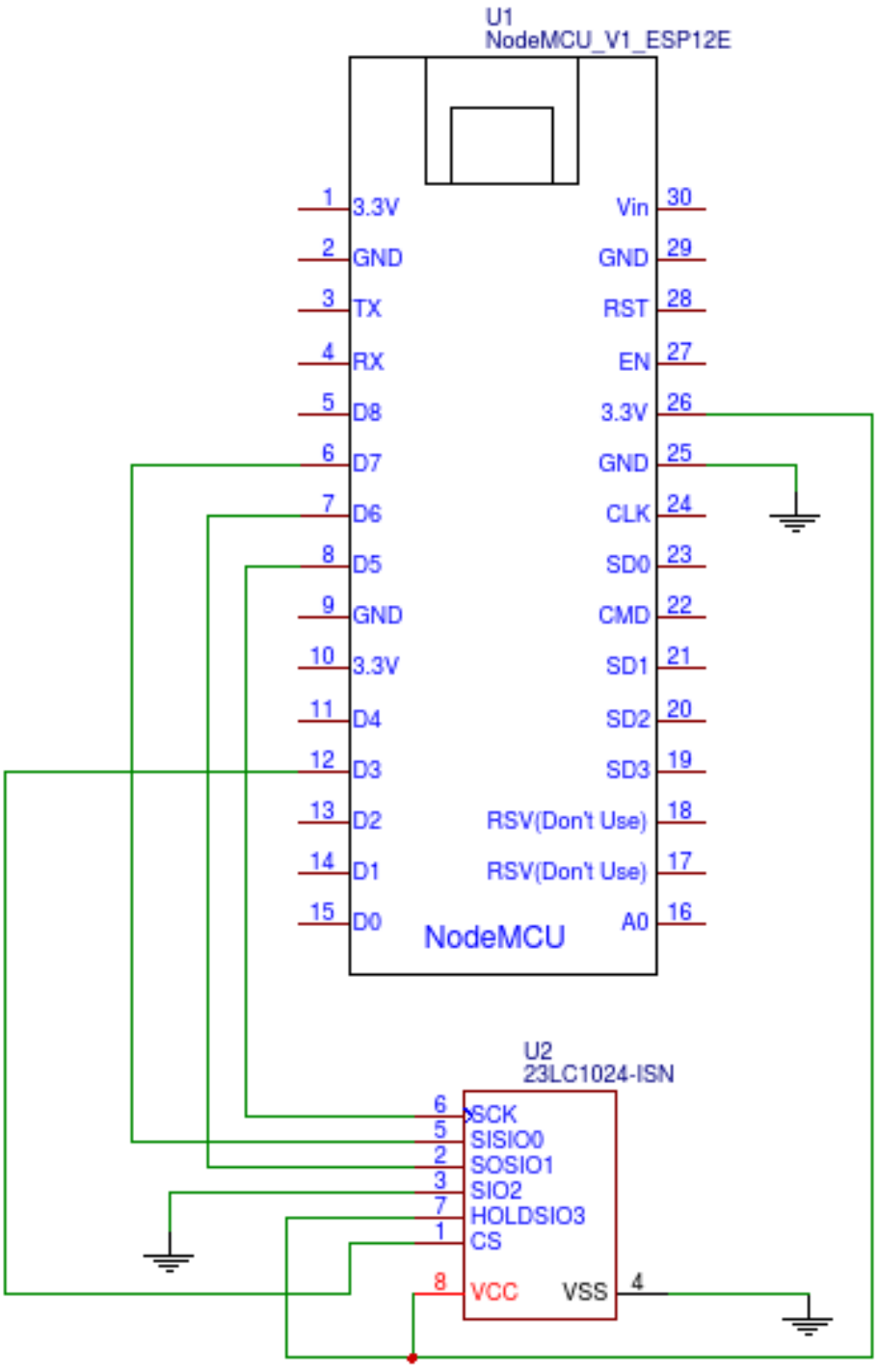


Figure 2.4 – Principal scheme

2.16 Conclusion

This chapter provides a comprehensive overview of the components, software, and technical tools necessary for controlling sound codecs using the ESP8266 microcontroller. The chapter began with an introduction to the ESP8266Audio library, highlighting its role and significance in managing audio operations on the ESP8266 platform. Detailed instructions on setting up the ESP8266Audio environment were provided, ensuring that users can easily begin their projects. Various classes such as AudioInput, AudioFileSource, AudioGenerator, and AudioOutput were explained, emphasizing their functionalities and how they interact to facilitate audio processing.

The integration and usage of different Digital-to-Analog Converters (DACs), including the Adafruit I2S DAC and PCM5102 DAC, were explored. These components are crucial for converting digital audio signals into analog signals that can be played through speakers. The chapter covered the implementation of a software I2S Delta-Sigma DAC, which allows for music playback using minimal hardware—specifically, a single transistor and a speaker. This approach showcases the versatility and cost-effectiveness of the ESP8266 microcontroller in audio applications.

Practical solutions for common issues, such as high-pitched buzzing with the 1-T circuit and debugging the 1-T amp circuit, were provided, ensuring smoother development and deployment processes. Advanced topics such as SPDIF optical output and the use of external SPI RAM to increase buffer sizes were discussed, offering insights into enhancing audio performance and expanding the capabilities of the ESP8266.

The usage of SD cards with ESP8266Audio was covered, providing a method for handling large audio files and extending the storage capacity of the microcontroller.

The chapter touched upon the potential for porting the ESP8266Audio functionalities to other microcontrollers, highlighting the adaptability and scalability of the techniques discussed.

By addressing these key areas, the chapter equips developers with the knowledge and tools required to effectively control sound codecs using the ESP8266 microcontroller. From basic setup and input/output management to advanced troubleshooting and

performance enhancement, the comprehensive coverage ensures that readers are well-prepared to tackle a variety of audio-related projects. The versatility and robustness of the ESP8266 platform, combined with the detailed guidance provided, pave the way for innovative and efficient audio applications.

3 IMPLEMENTATION OF THE SOFTWARE AND TECHNICAL TOOL FOR CONTROLLING SOUND CODECS BASED ON THE ESP8266 MICROCONTROLLER

3.1 Software implementation

Software implementation includes the st of header files for implementation of the software and technical tool for controlling sound codecs based on the esp8266 microcontroller. Let us consider them.

3.1.1 AudioFileSource header file implementation

AudioFileSource is a base class of an input "file" to be used by AudioGenerator.

The provided code defines the AudioFileSource class, which serves as an abstract base class for managing audio file sources. This class provides a set of virtual functions that can be implemented by derived classes to handle various audio file operations.

The key components and functionality of the AudioFileSource class are as follows:

1. Constructors and Destructors:

- AudioFileSource(): Default constructor.
- ~AudioFileSource(): Virtual destructor.

2. Audio File Operations:

- open(const char *filename) opens the specified audio file.
- read(void *data, uint32_t len) reads data from the open audio file.
- readNonBlock(void *data, uint32_t len) reads data from the open audio file in a non-blocking manner.
- seek(int32_t pos, int dir) seeks to a specific position in the open audio file.
- close() closes the open audio file.
- isOpen() checks if the audio file is currently open.
- getSize() retrieves the size of the open audio file.
- getPos() retrieves the current position in the open audio file.

- loop() checks if the audio file should be played in a loop.

3. Callback Registration:

- RegisterMetadataCB(AudioStatus::metadataCBFn fn, void *data) registers a callback function to handle audio metadata events.

- RegisterStatusCB(AudioStatus::statusCBFn fn, void *data) registers a callback function to handle audio status events.

The AudioStatus class is used to manage the metadata and status callbacks for the AudioFileSource class. It provides methods to register the callback functions and handle the corresponding events.

The header filerepresents the structure and relationships of the AudioFileSource class and its associated components. The diagram includes detailed explanations for each of the class's methods, highlighting their purpose, parameters, and return values.

The diagram is organized into a single class diagram, with the AudioFileSource class as the main component. The AudioStatus class is shown as a separate component, with a composition relationship to the AudioFileSource class, indicating that the AudioFileSource class uses the AudioStatus class to manage its callbacks.

The notes attached to the various methods provide concise and informative explanations to help the user understand the functionality of each component within the AudioFileSource class.

Here is a detailed header filewith explanations for the provided code:The provided code consists of several classes that implement the AudioFileSource interface, each with its own functionality and purpose.

AudioFileSourceBuffer class provides a buffered interface for reading data from an underlying AudioFileSource. It maintains an internal buffer to optimize read operations and handle potential underflow conditions. The buffer can be allocated dynamically or provided as a pre-allocated memory buffer by the application.

The class supports seeking within the buffer and the underlying AudioFileSource.

It also provides methods to check the fill level of the buffer and loop the underlying AudioFileSource.

AudioFileSourceFS class provides an implementation of the AudioFileSource interface for reading files from a filesystem (e.g., SPIFFS or FAT filesystem).

It encapsulates the filesystem-specific operations, such as opening, reading, seeking, and closing files.

The class can be initialized with a specific filesystem instance or a filename to open the file directly.

AudioFileSourceFunction generates audio data on-the-fly based on user-provided callback functions.

It sets up a WAV header with the specified audio parameters (duration, channels, sample rate, bit depth) and generates the audio data using the provided callback functions.

The class supports adding one or more callback functions, either unique for each channel or shared across channels.

It provides methods to read the generated audio data, seek within the audio, and retrieve information about the generated audio.

AudioFileSourceHTTPStream class provides an implementation of the AudioFileSource interface for reading audio data from an HTTP stream.

It encapsulates the HTTP client functionality, including opening the stream, reading data, and handling disconnections.

The class supports reconnecting to the stream if a disconnection occurs, with configurable retry attempts and delay.

It provides methods to read data from the stream, both in blocking and non-blocking modes, as well as to retrieve information about the stream, such as its size and current position.

The header file provides a detailed visual representation of the logic and functionality within each of these classes. The subgraphs represent the different classes, and the nodes within each subgraph correspond to the key methods and operations performed by the class.

The explanations provided alongside the diagram elements help to understand the purpose and behavior of each component of the code, making it easier for the user to

comprehend the overall structure and functionality of the audio file source implementations.

The flowchart of the algorithm is shown in Figure 3.1.

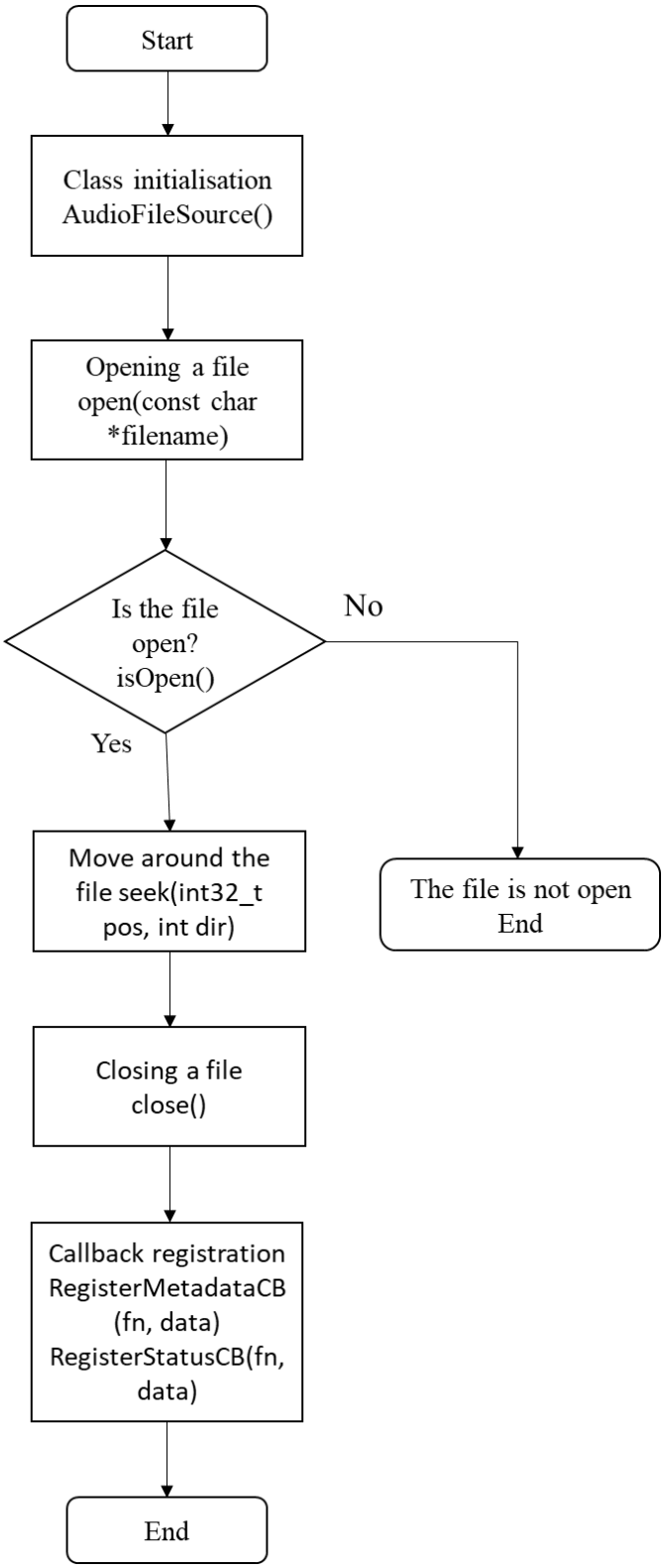


Figure 3.1 - The flowchart of the algorithm

3.1.2 AudioFileSourceBuffer header file implementation

The code implements an AudioFileSourceBuffer class which acts as a buffer for an audio file meant for playback. It provides efficient access to the audio data by reducing the number of reads required from the underlying audio source.

The class offers two constructors for flexibility:

One that takes an AudioFileSource object and a buffer size as parameters. This likely creates a new buffer to hold the audio data.

Another that takes an AudioFileSource object, a pointer to an existing buffer and a buffer size as parameters. This allows you to use a pre-allocated buffer.

The class includes a destructor to properly deallocate any memory it allocated for the buffer.

The functions provided by the class are:

- Seek allows you to move the playback position within the audio file based on different seek modes (e.g. moving a specific number of bytes forward or backward).
- Close closes the underlying audio source, likely releasing any resources associated with it.
- isOpen checks if the underlying audio source is still open.
- getSize gets the size of the entire audio file.
- getPos gets the current playback position within the audio file.
- getFillLevel gets the number of bytes currently filled in the buffer. This is useful to determine how much data is available for playback without needing to read from the source.
- read function for data access. It prioritizes reading from the buffer first for efficiency. If the buffer doesn't have enough data to satisfy the read request, it then reads from the underlying audio source.
- Fill function proactively tries to fill the buffer with data from the underlying audio source when the buffer isn't full. This helps to avoid stalls during playback when the buffer runs out of data.

- Loop enables looping playback on the underlying audio source. This is useful for creating continuous playback of the audio file.

The flowchart of the algorithm is shown in Figure 3.2.

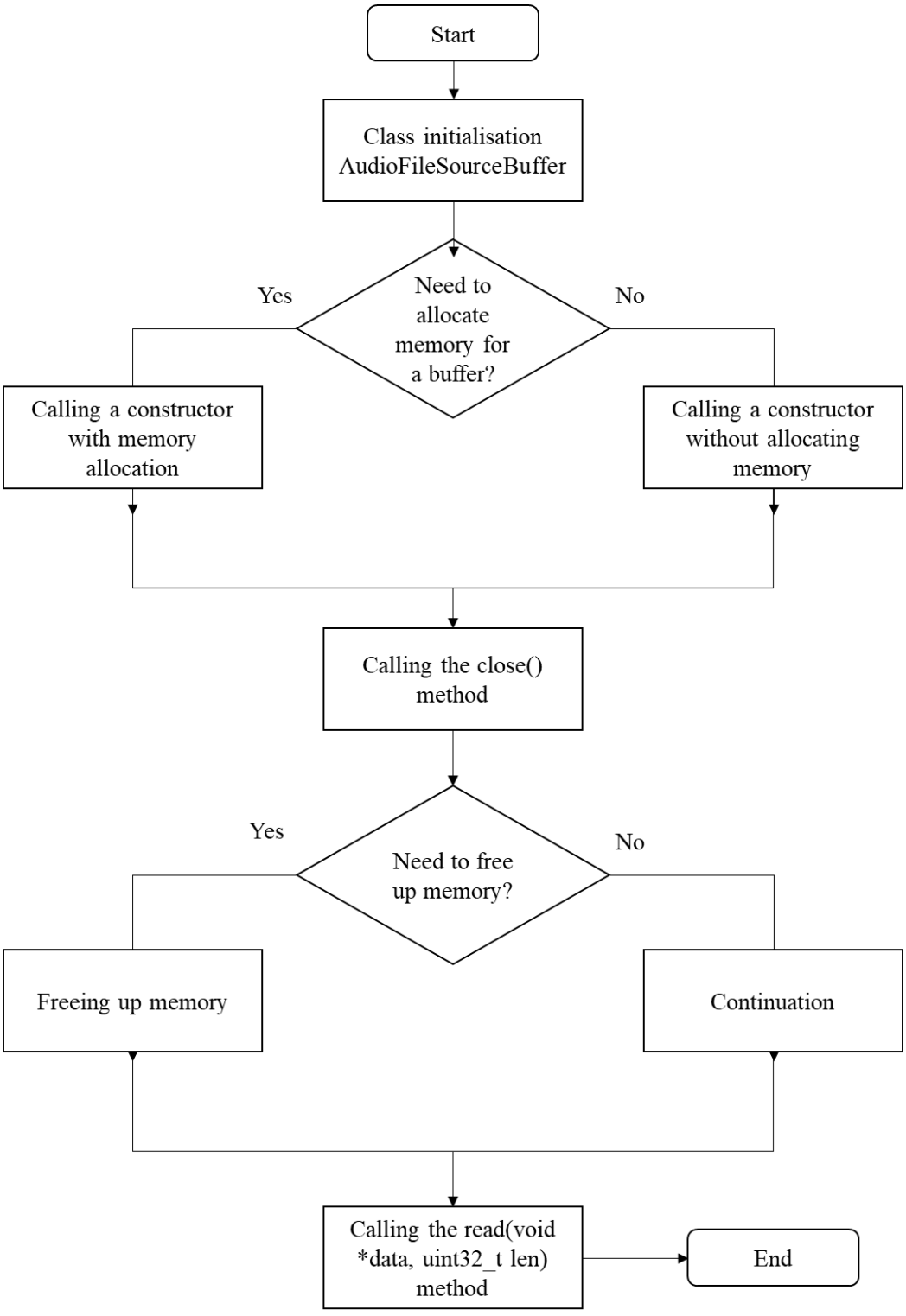


Figure 3.2 - The flowchart of the algorithm

3.1.3 AudioFileSourceBuffer header file implementation

This code defines a header file for the AudioFileSourceBuffer class, which acts as a buffer for audio data during playback. The header file serves two main purposes:

Constructor takes an AudioFileSource object and a buffer size as parameters. This likely creates a new buffer to hold the audio data.

Another takes an AudioFileSource object, a pointer to an existing buffer, and a buffer size as parameters. This allows you to use a pre-allocated buffer, providing more flexibility in memory management.

The destructor (~AudioFileSourceBuffer) is responsible for proper deallocation of any memory the class allocates for the buffer, ensuring efficient resource management.

Overridden virtual functions inherit from the parent class AudioFileSource and are likely overridden to provide buffering functionality around the original implementation.

Read function prioritizes reading data from the buffer first for efficiency. If the buffer doesn't have enough data to satisfy the read request, it then reads from the underlying audio source.

Seek allows you to move the playback position within the audio file based on different seek modes (e.g. moving a specific number of bytes forward or backward).

Close closes the underlying audio source, likely releasing any resources associated with it.

isOpen checks if the underlying audio source is still open.

getSize gets the size of the entire audio file.

getPos gets the current playback position within the audio file.

Loop enables looping playback on the underlying audio source. This is useful for creating continuous playback of the audio file.

New member function retrieves the number of bytes currently filled in the buffer. This is useful to determine how much data is available for playback without needing to read from the source, improving efficiency.

fill function is declared as private, meaning it can only be accessed by the AudioFileSourceBuffer class itself. It's likely responsible for refilling the buffer from the source when the buffer runs low on data, preventing playback stalls.

Member variables store information about the buffer, the underlying source, and the playback state. They include:

Src is a pointer to the AudioFileSource object representing the underlying audio source.

buffSize is the size of the buffer in bytes.

Buffer is a pointer to the buffer memory location.

deallocateBuffer is a flag indicating whether the class is responsible for deallocating the buffer memory.

writePtr is a variable tracking the write position within the buffer (where new data is written).

readPtr is a variable tracking the read position within the buffer (where data is read from).

Length is the number of bytes currently filled in the buffer.

Filled is a flag indicating whether the buffer is currently full.

By encapsulating these details within a header file, the code promotes better organization, maintainability, and reusability of the AudioFileSourceBuffer class. Other parts of the code can interact with the class through the public interface without needing to know the internal implementation details.

3.1.4 AudioFileSourceFATFS header file implementation

AudioFileSourceFS defines a header file for the AudioFileSourceFATFS class, specifically designed to work with ESP32 boards for playing audio files stored on a FAT filesystem (commonly used in SD cards). It promotes code organization, reusability, and separation of concerns.

The class inherits from AudioFileSourceFS, which likely provides a foundation for handling audio files from various file systems. This promotes code reuse and reduces redundancy by inheriting common functionalities like read, seek, and close for audio file manipulation.

The first constructor (AudioFileSourceFATFS()) is a default constructor with no arguments. It likely serves as a base case for creating an AudioFileSourceFATFS object.

The second constructor (AudioFileSourceFATFS(const char *filename)) takes a filename as a parameter but has a different behavior compared to a typical constructor. It initializes the parent class (AudioFileSourceFS) but avoids calling the parent class's open function directly with the filename. This suggests a potential design choice to avoid unintended behavior in the parent class's open implementation.

Overridden open function: This function plays a crucial role in ensuring successful playback from the SD card.

FAT Filesystem Mounting Check (!FFat.begin()) verifies if the FAT filesystem is already mounted. If not, it attempts to mount the filesystem using the FFat.begin() function.

Error Handling (!FFat.begin()) logs an error message using audioLogger->printf_P likely indicating an issue with the SD card or filesystem. It then returns false, signaling an error to the calling code.

Successful Mounting (FFat.begin()) calls the parent class's AudioFileSourceFS::open(filename) to perform the actual file opening process using the provided filename.

In essence, this class acts as an adapter for the AudioFileSourceFS class, specifically tailored to handle audio files stored on an SD card's FAT filesystem on ESP32 boards. It ensures proper filesystem mounting before attempting to open the audio file, enhancing robustness and preventing potential errors during playback.

The flowchart of the algorithm is shown in Figure 3.3.

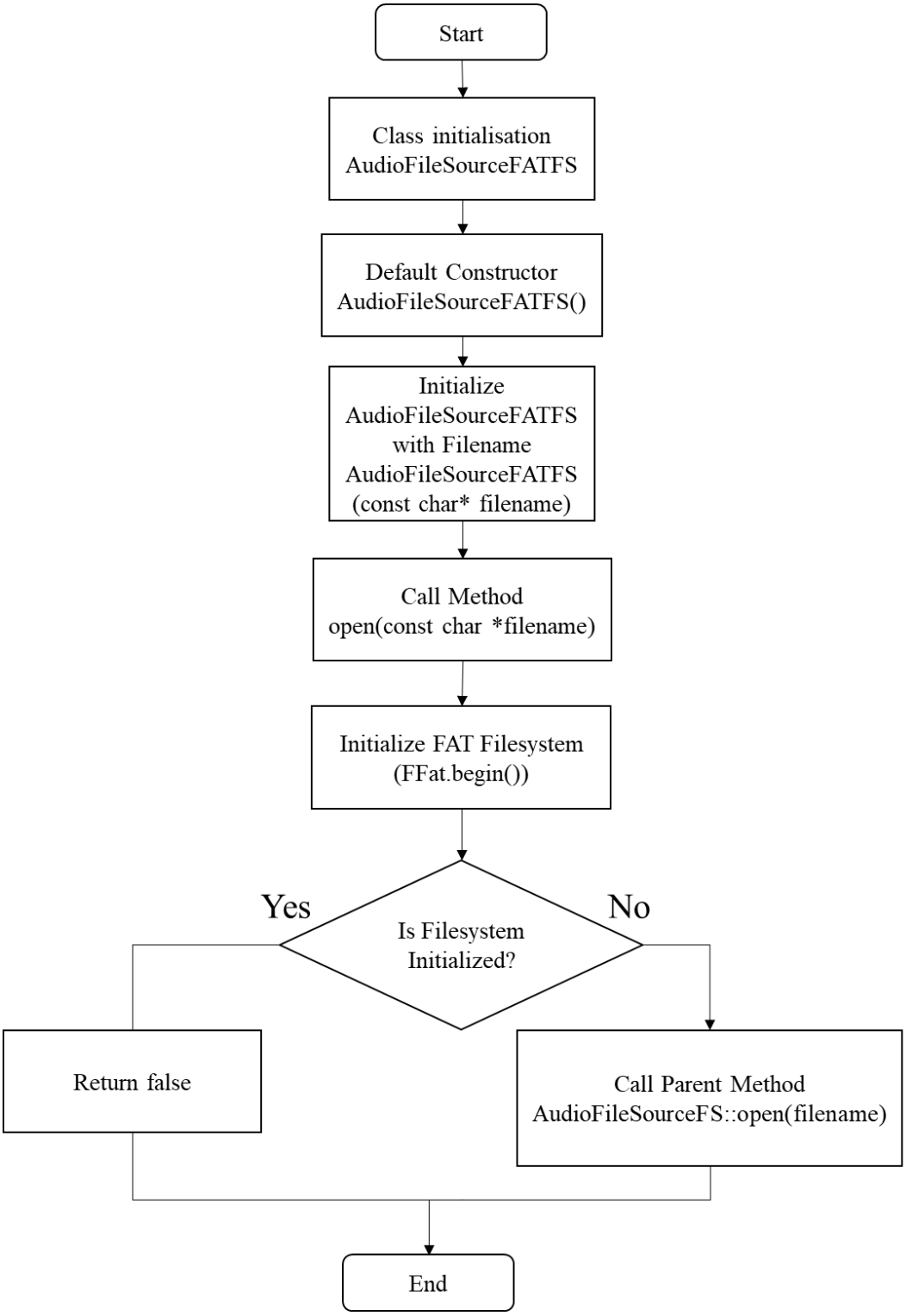


Figure 3.3 - The flowchart of the algorithm

3.1.5 AudioFileSourceHTTPStream header file implementation

AudioFileSourceHTTPStream header file defines a header file for the AudioFileSourceHTTPStream class, designed to play audio streams delivered over HTTP

protocols (like MP3 streams) on ESP32 or ESP8266 boards using the Arduino framework. It promotes code modularity, reusability, and stream playback functionality.

udioFileSourceHTTPStream class inherits from the AudioFileSource class, likely providing a foundation for handling audio data in a general way. This promotes code reuse by inheriting functionalities like read, seek, and close which can be adapted for the specific context of HTTP streams. Inherited functions might require re-implementation to work with continuous streams that lack a fixed size and traditional seeking capabilities.

Member variables of the class are:

- Client is a WiFiClient object, likely used for managing the low-level network connection to the HTTP server hosting the audio stream.
- http is a HTTPClient object, used for sending HTTP requests and receiving data chunks from the server in a higher-level manner compared to WiFiClient. It simplifies the process of communicating with the server using HTTP protocol.
- Pos is an integer variable that might be intended to keep track of the current playback position within the stream. However, unlike traditional audio files with a defined size, HTTP streams are continuous and seeking to specific points might not be possible. The value of pos might be used for informational purposes or limited seeking capabilities within the stream.
- Size is aAn integer variable that might store the reported size of the stream if available from the server. However, unlike downloadable files, HTTP streams are typically open-ended and their total size might not be known beforehand. The value of size could be used if reliable size information is provided by the server.
- reconnectTries is an integer used to configure the number of attempts to re-establish the connection to the server in case the connection is lost during playback. This helps with robustness in case of network interruptions.
- reconnectDelayMs is an integer specifying the delay (in milliseconds) between reconnect attempts. It provides a way to control the frequency of retries and avoid overwhelming the server with too many connection requests in a short time.

- saveURL is a character array to store the URL of the audio stream. This is likely used to reference the stream location and potentially manage multiple streams or reconnect attempts.

Constructor FileSourceHTTPStream() is for creating an empty AudioFileSourceHTTPStream object, potentially initializing member variables to default values.

Constructor AudioFileSourceHTTPStream(const char *url) takes a URL as input and might be responsible for several tasks:

Storing the URL in the saveURL member variable for future reference.

Creating or initializing the WiFiClient and HTTPClient objects for network communication.

Potentially opening the HTTP connection to the provided URL and initiating the stream. Depending on the server implementation, this might involve sending specific HTTP requests or headers to start streaming the audio data.

Potentially buffering some initial data from the stream to avoid immediate delays during playback.

Destructor (~AudioFileSourceHTTPStream) handles proper cleanup of resources associated with the network connection and the HTTPClient object. It might close the connection, release any allocated memory, and ensure the resources are freed to prevent memory leaks.

Overridden virtual functions provide implementations specific to handling HTTP audio streams, likely inheriting some behavior from the parent class AudioFileSource but adapting it to the characteristics of continuous streams.

Function open(const char *url) is responsible for opening the HTTP connection to the provided URL using the WiFiClient and HTTPClient objects. It might initiate the process of retrieving audio data from the server.

Functionread(void *data, uint32_t len) trieves data from the stream and fills the provided buffer with audio data. It might involve using the HTTPClient object to read data chunks from the server in a loop until the requested amount of data (len) is fulfilled.

Additionally, it might handle potential issues like connection errors or buffer underruns (when not enough data is available to fill the buffer). Reconnection logic might be implemented here to automatically attempt to re-establish a connection if it's lost during playback.

Function readNonBlock(void *data, uint32_t len) provides a way to read data in a non-blocking manner, potentially useful for avoiding delays in the playback process.

3.1.6 AudioFileSourceFS header file implementation

AudioFileSourceFS header file was esigned to work with various file systems (FS) for audio playback on Arduino boards. It provides a foundation for handling audio files stored on different storage media like SD cards or internal flash memory. It ensures the class is only included once in a project, preventing errors from multiple inclusions.

The class inherits from AudioFileSource, which likely provides a foundation for handling audio data in general. This promotes code reuse by leveraging existing functionalities for error handling, basic read/write operations, and potential base class implementations for close and isOpen.

The constructor AudioFileSourceFS(fs::FS &fs) takes a reference to an fs::FS object as a parameter. This object likely represents the specific file system being used (e.g., SD card filesystem, internal flash filesystem).

The constructor (AudioFileSourceFS(fs::FS &fs, const char *filename)) takes both the file system reference and a filename as parameters. It likely attempts to open the specified audio file using the provided file system.

Overridden member functions provide the functionalities for managing audio files on a file system. open(const char *filename) function is responsible for opening the audio file specified by the filename parameter using the file system referenced by the filesystem member variable. It likely performs operations like opening the file, checking for errors, and potentially storing the opened file handle internally. read(void *data, uint32_t len) function allows reading audio data from the opened file. It likely reads a specified number

of bytes (len) into the provided buffer (data). seek(int32_t pos, int dir) function enables seeking within the audio file based on the specified position (pos) and direction (dir). The implementation might vary depending on the underlying file system's capabilities. close() function likely closes the previously opened audio file, releasing any resources associated with it. isOpen() function checks if a file is currently open (f member variable) and returns a boolean value indicating the open state. getSize() function might retrieve the size of the audio file from the file system, potentially using the file system's functionalities to determine the file size. getPos() function retrieves the current read position within the opened file using the f.position() method of the fs::File object. Filesystem is areference to an fs::FS object representing the specific file system being used. f: An fs::File object stores the handle to the opened audio file. This is used for subsequent read/seek operations on the file. By encapsulating these functionalities within a header file, the code promotes better organization, maintainability, and reusability of the AudioFileSourceFS class. Other parts of the code can interact with the class through the public interface without needing to know the low-level details of interacting with different file systems.

The flowchart of the algorithm is shown in Figure 3.4.

3.1.7 AudioFileSourceFunction.cpp

Audio output generator can generate WAV file data from function.

This code defines a class named AudioFileSourceFunction that generates audio data on the fly based on user-provided functions and stores it in a WAV container format. It provides functionalities to control the generated audio data and offers an interface compatible with the AudioFileSource class.

Constructor (AudioFileSourceFunction) takes arguments specifying audio properties.

Sec is a duration of the audio in seconds (float).

Channels is the number of audio channels (e.g., mono = 1, stereo = 2).

sample_per_sec is a sample rate (number of samples per second).

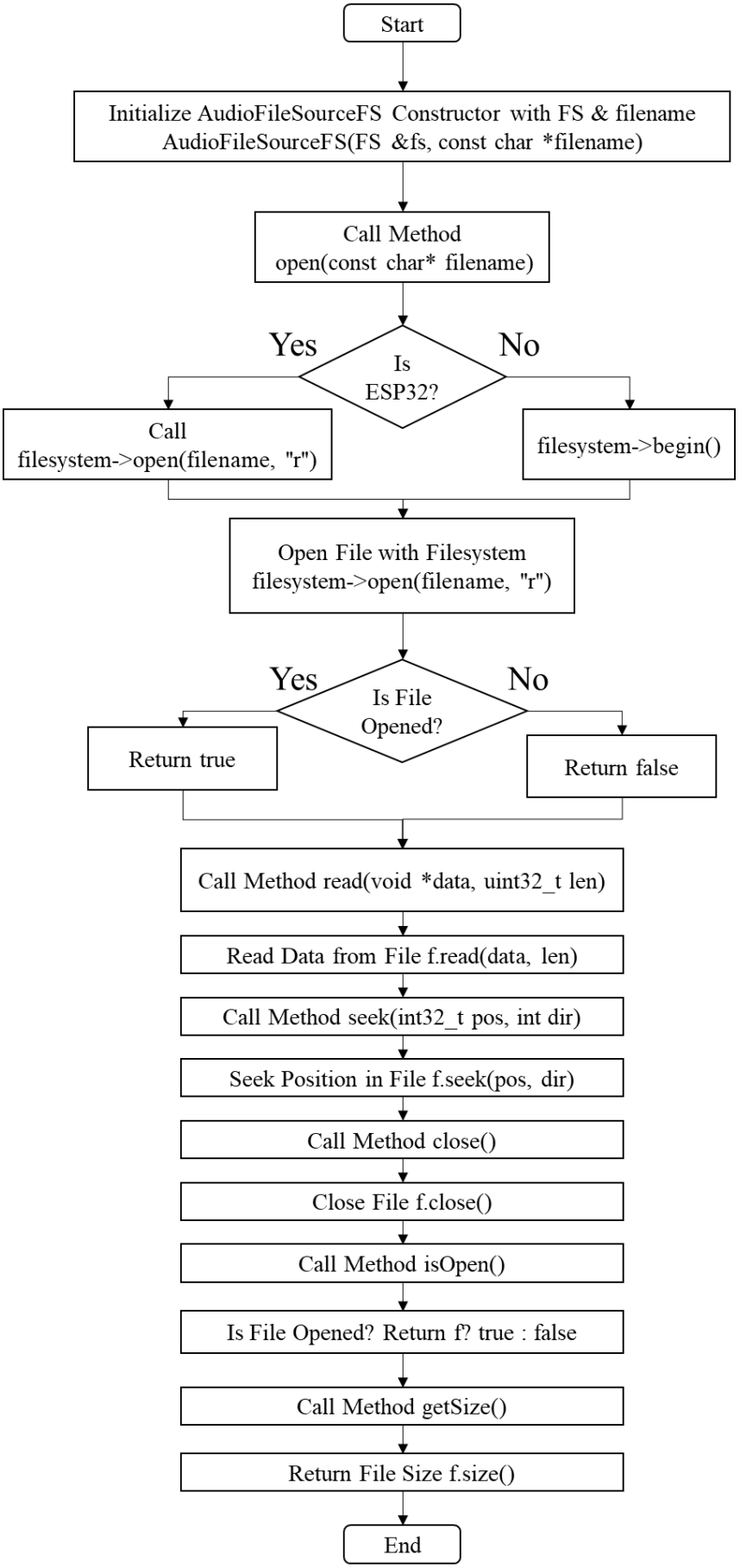


Figure 3.4 - The flowchart of the algorithm

bits_per_sample is a number of bits per sample (determines audio quality).

It calculates the total audio data size in bytes based on the provided parameters, and creates a WAV header structure (wav_header) and fills it with the corresponding values for chunk sizes, format details (channels, sample rate, etc.) based on the input arguments.

Let us consider member variables.

Funcs is a vector to store user-provided functions (likely lambdas or function pointers) that will be used to generate audio data for each channel.

Pos keeps track of the current read position within the audio data.

Size stores the total size of the audio data (including header).

is_ready is a flag indicating if the audio data generation is complete and ready for reading.

is_unique is a flag indicating if the provided functions are unique for each channel (otherwise the first function will be used for all channels).

Destructor (~AudioFileSourceFunction) calls the close function to ensure proper cleanup.

read(void* data, uint32_t len) reads audio data from the in-memory WAV container and copies it to the provided buffer (data) respecting the requested length (len).

For the header, it directly copies bytes from the pre-generated wav_header structure.

For audio data, it calculates the current time based on the read position and calls the appropriate user-provided function (funcs[channel]) to generate a sample value. This is done for each channel.

Based on the number of bits per sample, it converts the floating-point sample value to the corresponding integer representation and stores it in the output buffer.

Updates the pos variable to track the amount of data read.

Returns the number of bytes actually read, which might be less than the requested length if the end of the data is reached.

seek(int32_t pos, int dir) function allows seeking within the generated audio data based on the specified position (pos) and seek mode (dir).

SEEK_SET seeks from the beginning of the audio data.

SEEK_CUR seeks relative to the current position.

It performs validations to ensure the seek position is within the valid range of the audio data.

Updates the pos member variable to reflect the new seek position.

close() resets member variables to their initial states, effectively closing the audio source and releasing any associated resources.

isOpen() returns the is_ready flag indicating if the audio data generation is complete.

getSize() returns the total size of the audio data (including header).

getPos() returns the current read position within the audio data.

Overall, the AudioFileSourceFunction class provides a way to generate audio data in memory using user-defined functions and serves as a virtual audio file source compatible with the AudioFileSource class. This allows for creating dynamic audio experiences without the need for pre-recorded audio files.

The flowchart of the algorithm is shown in Figure A.1 of Annex A.

3.1.8 AudioFileSourceFunction header file implementation

Audio output generator can generate WAV file data from function. The flowchart of the algorithm is shown in Figure A.2 of Annex A.

The AudioFileSourceFunction class facilitates in-memory generation of audio data based on user-provided functions and presents an interface compatible with the AudioFileSource class. This enables creating dynamic audio experiences on the fly without relying on pre-recorded audio files.

WAV Header Structure (WavHeader) is nested within AudioFileSourceFunction, this struct meticulously defines the layout of a WAV header.

It utilizes byte arrays (char arrays) to represent various header fields, including:

RIFF Chunk identifies the file as a WAV container.

Format Chunk specifies details about the audio format (channels, sample rate, bit depth).

Data Chunk indicates the size and location of the actual audio sample data within the file.

Integer Conversion Unions (Uint8AndIntX) unions (where X represents 8, 16, or 32) provide a mechanism for efficient conversion between signed integer (intX) and unsigned integer (uintX) representations for various data sizes (8, 16, or 32 bits).

This proves beneficial when dealing with byte packing for sample data, ensuring correct manipulation of numerical values based on the chosen bit depth.

Function Pointer Definition (callback_t) acts as an alias for std::function<float(float)>. In essence, it represents a function pointer or lambda expression.

Member Variables are:

- std::vector<callback_t> that stores the user-supplied audio generation functions.
- Pos keeps track of the current read position within the in-memory audio data.
- Size stores the total size of the generated audio data, encompassing the header.
- is_ready is a flag indicating whether the audio data generation process is complete and ready for reading.
- is_unique is a flag indicating if all channels utilize the same audio generation function.

Constructor (AudioFileSourceFunction) takes arguments specifying audio properties like duration (seconds), number of channels (mono/stereo), sample rate, and bit depth per sample.

Initializes the wav_header structure with the provided parameters, meticulously setting the header fields to accurately reflect the chosen audio format.

Destructor (~AudioFileSourceFunction) ensures proper resource management by calling the close function during object destruction.

addAudioGenerators functions serve the purpose of adding user-provided functions (either single or multiple) that will be responsible for generating audio samples.

The templated version (addAudioGenerators(const F& f, Fs&&... fs)) empowers the addition of multiple functions using variadic arguments, offering flexibility in defining audio generation behavior.

They perform validations to ensure the number of provided functions aligns with the number of channels or is limited to a single function (used for all channels).

Based on the validation results, they update the is_ready and is_unique flags to reflect the configuration's validity. Additionally, they might clear the funcs vector if an invalid configuration is encountered.

Reading audio data from the in-memory WAV container.

Seeking within the generated audio data based on a specified position and seek mode. Closing the audio source and releasing associated resources.

Checking if the audio data generation is complete (ready for reading).

Retrieving the total size of the generated audio data (including header).

Obtaining the current read position within the audio data. In essence, the AudioFileSourceFunction class and its supporting functionalities empower the creation of dynamic audio experiences on embedded systems like Arduino boards. It provides a mechanism to generate audio data in real-time using user-defined functions, eliminating the need for pre-recorded audio files.

3.1.9 AudioFileSourceHTTPStream.cpp

The AudioFileSourceHTTPStream class facilitates playing audio streams delivered over HTTP on ESP32 and ESP8266 boards. It inherits from the AudioFileSource class, adhering to a common interface for handling audio sources. This class empowers the creation of audio playback functionalities that leverage HTTP streams as the audio source,

enabling playback of remote audio content without storing the audio data locally on the device. The flowchart of the algorithm is shown in Figure A.3 of Annex A.

HTTP Client (http) is an instance of the HTTPClient class is utilized to manage the HTTP connection with the remote server hosting the audio stream.

It provides methods for opening connections, sending requests (like GET in this case), receiving responses, and accessing the response data.

The class implements mechanisms to establish and maintain the HTTP connection throughout the playback process.

It checks the response code after sending the GET request to ensure a successful connection (HTTP code 200 - OK). If an error occurs, it closes the connection, reports the error through a callback (cb.st), and returns an error code.

It incorporates retry logic to handle potential connection drops during playback. Upon disconnection, it attempts to re-establish the connection a specified number of times (reconnectTries) with a delay between attempts (reconnectDelayMs). Informative messages are provided through the callback during the reconnection process.

The read and readNonBlock functions provide mechanisms to retrieve audio data from the established HTTP stream.

They handle potential situations where the requested data length exceeds the remaining data in the stream, adjusting the read length to avoid exceeding the stream's boundaries.

The readInternal function serves as the core for data reading. It implements blocking or non-blocking reads based on the provided flag (nonBlock). Blocking reads wait for data to become available using yield, while non-blocking reads check for immediate availability using stream->available().

Once data is available, it's read from the stream using stream->read and stored in the provided buffer.

The pos member variable is updated to track the amount of data read from the stream.

The seek function is currently not implemented (audioLogger->printf_P(...)) as seeking within an HTTP stream might not be feasible due to the nature of streaming data delivery. It always returns false to indicate that seeking is not supported.

The close function gracefully terminates the HTTP connection when the playback is complete or when an error occurs.

The isOpen function provides a way to check if the HTTP connection is currently active.

The getSize function, if the server provides the content length information, returns the total size of the audio stream. However, this might not always be available for HTTP streams.

The getPos function returns the current read position within the stream, indicating how much data has been read so far.

In essence, the AudioFileSourceHTTPStream class empowers the development of audio playback applications on ESP32 and ESP8266 boards that leverage HTTP streams as the audio source. It offers functionalities for connection management, error handling, data reading, and basic status checks, enabling the creation of dynamic and remote audio playback experiences.

3.1.10 AudioFileSourceHTTPStream header implementation

The AudioFileSourceHTTPStream class facilitates playing audio streams delivered over HTTP on ESP32 and ESP8266 boards. It inherits from the AudioFileSource class, adhering to a common interface for handling audio sources within the project. This class empowers the creation of audio playback functionalities that leverage HTTP streams as the audio source, enabling playback of remote audio content without storing the audio data locally on the device. The flowchart of the algorithm is shown in Figure A.4 and FA.5 of Annex A.

The code leverages conditional compilation (#ifdef and #else) to ensure the inclusion of the appropriate HTTP client library (HTTPClient for ESP32 or

ESP8266HTTPClient for ESP8266) based on the target board. This promotes code portability and maintainability across different development boards.

Member Variables are:

- client is a WiFiClient object, likely used for establishing the underlying network connection to the remote server hosting the audio stream.
- http is a HTTPClient object used for managing HTTP requests (like GET in this case) and responses. This object handles communication details with the server.
- Stream Position and Size Tracking:
- Pos tracks the current read position within the audio stream, indicating how much data has been read so far. This value is crucial for keeping track of playback progress.
- Size stores the total size of the audio stream, if available from the server's response. This might not always be provided, especially for live streams.
- reconnectTries configures the number of attempts to reconnect if the connection drops during playback. This mechanism helps to improve robustness in case of network issues.
- reconnectDelayMs sets the delay (in milliseconds) between reconnect attempts. This parameter allows for fine-tuning the reconnection behavior, balancing between persistence and efficiency.
- saveURL stores the URL of the opened audio stream for potential reuse. This can be helpful for scenarios where the same stream needs to be played back multiple times or for managing different playback sessions.

AudioFileSourceHTTPStream() is a default constructor, likely initializes member variables to default values or empty states.

AudioFileSourceHTTPStream(const char *url) takes a URL as an argument. It attempts to open the audio stream using the provided URL and establish the HTTP connection. This constructor simplifies stream opening for the user.

Destructor (~AudioFileSourceHTTPStream) ensures proper resource management, likely closing the HTTP connection when the object is destroyed. This helps to avoid resource leaks and potential issues.

Stream Management functions are:

- open(const char *url) opens the specified audio stream URL and establishes the HTTP connection with the remote server. This function is responsible for initiating the playback process.
- close() gracefully terminates the HTTP connection, releasing associated resources like network connections. This function is essential for stopping playback and cleaning up after use.
- isOpen() checks if the HTTP connection is currently active, indicating whether an audio stream is being played. This can be useful for implementing playback controls or monitoring the playback state.

Data Reading functions are:

- read(void *data, uint32_t len) reads audio data from the stream into the provided buffer (data) respecting the requested length (len). This function implements blocking read behavior, meaning it might wait until the requested amount of data is available before returning.
- readNonBlock(void *data, uint32_t len) attempts a non-blocking read, checking for data availability before proceeding. This function is useful in situations where immediate response is important, and waiting for data can introduce delays.

Status and Information:

- getSize() returns the total size of the audio stream, if available (might not always be provided by the server). This can be useful for estimating playback duration or displaying progress information.
- getPos() returns the current read position within the stream. This value reflects how much data has been read from the stream so far and can be used to track playback progress.

3.1.11 AudioFileSourceICYStream.cpp

AudioFileSourceICYStream code defines the AudioFileSourceICYStream class, which inherits from AudioFileSourceHTTPStream and facilitates playing audio streams delivered in the ICY (Icecast) format on ESP32 and ESP8266 boards. ICY is a streaming protocol commonly used for internet radio broadcasts. This class enables the development of audio playback applications that leverage ICY streams as the audio source.

Member Variables are:

- icyMetaInt stores the metadata interval value retrieved from the ICY header, indicating the frequency of metadata updates (e.g., station name, song information) embedded within the stream. This value specifies the number of bytes between metadata blocks.
- icyByteCount tracks the number of bytes read since the last encountered ICY metadata block. This value is crucial for determining potential overlaps between requested data and metadata boundaries during the reading process.

Public Member Functions are:

- AudioFileSourceICYStream() is a default constructor, likely initializes member variables to default values or empty states.
- AudioFileSourceICYStream(const char *url) is a constructor that takes a URL as an argument. It attempts to open the ICY stream using the provided URL, establish the HTTP connection with the server, and parse the initial headers to extract relevant information.
- open(const char *url) opens the specified ICY stream URL using HTTP.

It sends additional headers in the HTTP request (Icy-MetaData: 1) to instruct the server to include metadata information within the stream.

It collects specific headers (icy-metaint, icy-name, icy-genre, icy-br) using the collectHeaders function from the underlying HTTP client library. These headers might contain station name, genre, bitrate, and the crucial icy-metaint value indicating the metadata interval.

It parses the retrieved headers to extract the icyMetaInt value and potentially populate station name, genre, and bitrate (commented out sections // cb.md(...) likely using a callback function to notify the application about this information). Stores the content length (size) if provided by the server in the response headers.

Destructor (~AudioFileSourceICYStream) inherited from AudioFileSourceHTTPStream. Ensures proper resource management, likely closing the HTTP connection when the object is destroyed.

Private Member Function is readInternal(void *data, uint32_t len, bool nonBlock), and it overrides the readInternal function from the parent class to handle ICY metadata headers. The core logic focuses on handling potential overlaps between the requested data length (len) and ICY metadata boundaries.

It ensures that reads don't accidentally cross over into an ICY metadata block, which would disrupt the audio stream.

If a potential overlap is detected between requested data and a metadata boundary:

It reads data up to the metadata boundary, respecting the requested len but ensuring it doesn't exceed the boundary.

It parses the encountered ICY metadata block to extract information like "StreamTitle" (commented out section // cb.md(...), likely using a callback function to notify the application). This parsing might involve reading additional bytes to determine the complete metadata block size and then extracting the relevant details.

It skips the remaining bytes of the metadata block to avoid delivering metadata as part of the audio data.

Finally, it performs the actual data read from the stream using the remaining len after accounting for potential metadata handling.

It updates icyByteCount to track the total bytes read since the last encountered ICY metadata block. This value is essential for accurately calculating potential overlaps in subsequent read calls.

In essence, the AudioFileSourceICYStream class extends the AudioFileSourceHTTPStream functionality to handle ICY format audio streams. It

incorporates logic to parse and handle ICY metadata headers while ensuring seamless data reading without being interrupted by metadata boundaries. This enables the development of robust and informative ICY stream playback applications on ESP32 and ESP866 boards.

3.1.12 AudioFileSourceICYStream.h

AudioFileSourceICYStream.h code snippet unveils the header file declaration for the AudioFileSourceICYStream class, empowering the development of audio playback applications that handle ICY (Icecast) format streams on ESP32 and ESP8266 boards. Let's delve into the intricacies of this code:

<Arduino.h> provides fundamental functionalities for interacting with Arduino hardware and libraries.

Conditional HTTP Client Library Headers are:

- <HTTPClient.h> (for ESP32): Supplies classes and functions for managing HTTP requests and responses on the ESP32 board.
- <ESP8266HTTPClient.h> (for ESP8266) offers similar functionalities tailored for the ESP8266 board's networking capabilities.
- "AudioFileSourceHTTPStream.h" includes the declaration of the parent class AudioFileSourceHTTPStream. This class establishes the foundation for handling audio streams delivered over HTTP, providing essential functionalities for connection management, error handling, data reading, and status checks.

AudioFileSourceICYStream defines a class named AudioFileSourceICYStream that inherits from the AudioFileSourceHTTPStream class. Inheritance creates a parent-child relationship, where the child class (AudioFileSourceICYStream) inherits the functionalities of the parent class (AudioFileSourceHTTPStream) and can add specific behaviors for handling ICY streams.

Member Functions are:

- AudioFileSourceICYStream() is a default constructor, likely responsible for initializing member variables to default values or empty states.

- AudioFileSourceICYStream(const char *url) is a constructor and it takes a URL as an argument. It's probable that it initializes the object and attempts to open the ICY stream using the provided URL, establishing the necessary connection and potentially parsing initial headers.

- ~AudioFileSourceICYStream() is ae destructor, likely invoked when the object goes out of scope or is explicitly deleted. It's essential for performing cleanup tasks, such as closing the HTTP connection associated with the stream, to avoid resource leaks and potential issues.

- uint32_t readInternal(void *data, uint32_t len, bool nonBlock) function is inherited from the parent class (AudioFileSourceHTTPStream) and overridden in this child class (AudioFileSourceICYStream). It's the core mechanism for reading audio data from the ICY stream. The override keyword indicates that this function redefines the behavior of the parent class's function with the same name, specializing it to handle the intricacies of ICY format, which includes metadata headers interspersed within the audio data stream.

- int icyMetaInt stores the metadata interval value retrieved from the ICY header. This value specifies the frequency (number of bytes) between metadata updates (e.g., station name, song information) embedded within the stream.

- int icyByteCount

- keeps track of the number of bytes read since the last encountered ICY metadata block. It plays a crucial role in determining potential overlaps between requested data and metadata boundaries during the reading process. By keeping count of the bytes read, the readInternal function can avoid accidentally reading into metadata blocks, ensuring a seamless audio playback experience.

In essence, the AudioFileSourceICYStream.h header file lays the groundwork for a class specifically designed to handle ICY format audio streams. It inherits the functionalities of the AudioFileSourceHTTPStream class for general HTTP stream

management and adds specialized behavior to handle ICY metadata headers, enabling developers to create robust and informative ICY stream playback applications on ESP32 and ESP8266 boards.

3.1.13 AudioFileSourceID3.cpp

This header file introduces the AudioFileSourceID3 class, designed to seamlessly handle ID3 metadata parsing within audio files or streams. It inherits from the AudioFileSource class, likely providing functionalities for reading audio data from various sources. Here's a breakdown of the key elements, delving deeper into the inner workings:

AudioFileSourceID3 inherits from AudioFileSource. This establishes a parent-child relationship where AudioFileSourceID3 inherits core functionalities for reading audio data from the parent class (AudioFileSource). It then adds specific behavior for handling ID3 metadata parsing, making it a specialized class tailored for this purpose.

Member Variables are:

- src is a pointer stores a reference to the underlying AudioFileSource object. The AudioFileSourceID3 class acts as a wrapper, delegating most of the reading tasks to the underlying source while handling ID3 metadata parsing specifically. It essentially intercepts the data flow from the source, parses out the ID3 tags at the beginning (if present), and then allows the remaining audio data to pass through to the application.
- Checked is a boolean flag that serves as a gatekeeper, ensuring that the ID3 header is only checked and parsed once. This flag prevents redundant processing on subsequent reads.
- AudioFileSourceID3(AudioFileSource *src) initializes the object with a reference to the underlying AudioFileSource object. This creates an instance of the AudioFileSourceID3 class, ready to be used for reading audio data while handling ID3 metadata.

- ~AudioFileSourceID3() performs any necessary cleanup tasks associated with the object. In real-world scenarios, this might involve closing any temporary resources used during ID3 parsing or deallocating memory.
- read(void *data, uint32_t len) function is responsible for reading data. It acts as the entry point for the application's data requests. Here's a step-by-step breakdown of its logic.
- Initial Checkst checks the checked flag.

If checked is false (indicating the ID3 header hasn't been processed yet), it reads the initial 10 bytes from the underlying source to check for the ID3 header signature (e.g., "ID3").

If the signature matches the ID3 format, it parses the header to extract information like ID3 version, size, and the presence of unsynchronization bytes and extended header. This parsing helps determine how to interpret the following data.

It creates an AudioFileSourceUnsync object (explained later) to handle unsynchronization if necessary. Unsynchronization bytes are used in some ID3 versions to prevent misinterpretations during parsing.

It skips the extended header (if present) as it's not essential for extracting basic metadata like album, title, or artist.

It enters a loop to iterate through potential ID3 frames until the end of ID3 data is reached:

It reads the frame ID (typically 4 bytes) to identify the type of information the frame contains (e.g., album title, artist name).

It reads the frame size to determine how many bytes are dedicated to this particular frame's content.

It checks for frame compression (currently not supported by the provided code).

It reads the frame content based on the frame size and unsynchronization flag. This might involve reading additional bytes if unsynchronization is enabled.

It extracts relevant information like album, title, artist, year, etc., based on the frame ID. It potentially calls a callback function (cb.md) to notify the application about the

discovered metadata. This callback function likely resides in a separate part of the code and is responsible for handling the extracted metadata in a meaningful way (e.g., displaying it on the user interface).

Once all ID3 frames are processed, it signals the end of ID3 tags using the callback function (cb.md), informing the application that the following data is the actual audio content.

After the initial ID3 check (or if no ID3 header is found), it delegates the actual data reading to the underlying source (src->read) for the remaining data (audio content). This essentially bypasses the ID3 parsing logic and retrieves the raw audio bytes from the source.

3.1.14 AudioFileSourceID3

The AudioFileSourceID3.h header file unveils a class specifically designed to streamline ID3 metadata parsing within audio files or streams managed by the Arduino framework. It acts as a wrapper around another AudioFileSource object, transparently handling ID3 tag extraction while delegating the core audio data reading functionality. This approach simplifies application development by providing a unified interface for accessing both metadata and audio content.

When the read function of AudioFileSourceID3 is called, it intercepts the data stream from the underlying AudioFileSource object. During the initial read, it meticulously checks for the presence of an ID3 header, typically identified by the signature "ID3". If this signature is detected, the class springs into action, parsing the header to extract crucial information like the ID3 version, its overall size within the stream, and the existence of unsynchronization bytes (used in some versions to prevent misinterpretations during parsing). This initial parsing equips the class with the knowledge necessary to accurately interpret the following data stream.

With the header parsed, AudioFileSourceID3 enters a meticulous loop, iterating through potential ID3 frames. Each frame acts as a container, encapsulating a specific

type of metadata like album title, artist name, or track number. The class meticulously reads the frame ID (usually 4 bytes) to identify the kind of information it holds. It then retrieves the frame size, determining the number of bytes dedicated to this particular metadata entry. The class also checks for frame compression, although the provided code doesn't handle compressed frames currently. Finally, it extracts the actual frame content based on the frame size and unsynchronization flag, if applicable. This step might involve reading additional bytes to account for potential inconsistencies introduced by unsynchronization.

Once the frame content is retrieved, AudioFileSourceID3 leverages its knowledge of frame IDs to translate the raw bytes into human-readable information. For instance, if the frame ID corresponds to the album title, the class extracts the corresponding characters and likely employs a callback function (cb.md in the code) to notify the application about this discovered metadata. This callback function, residing in a separate part of the codebase, is responsible for handling the extracted metadata in a meaningful way, such as displaying it on a user interface or storing it for later use.

After meticulously processing all encountered ID3 frames, the class reaches the end of the ID3 data segment within the stream. To inform the application that the following data stream consists of the actual audio content, AudioFileSourceID3 likely utilizes the same callback function (cb.md) to signal the end of ID3 tags. This notification essentially acts as a handoff, letting the application know that it can now focus on receiving and processing the raw audio bytes.

Transparent Audio Data Reading: Following the initial ID3 check (or if no ID3 header is found), the read function seamlessly transitions into its core functionality: delegating the actual audio data reading to the underlying AudioFileSource object. This delegation ensures a continuous flow of data to the application. The application receives a unified stream, with ID3 metadata (through the callback function) seamlessly integrated at the beginning (if present), followed by the raw audio bytes representing the audio content itself.

3.2 Conclusion

The chapter on the implementation of software and technical tools for controlling sound codecs based on the ESP8266 microcontroller provides a comprehensive and detailed guide to the development process. This section highlights the practical steps taken to create and manage various audio file sources through the implementation of specific header files and associated C++ files. Here are the key takeaways from the chapter.

The chapter systematically covers the development and implementation of numerous components necessary for handling audio files. These include source files for various audio formats and data streams, such as FATFS, HTTP streams, ICY streams, and ID3 metadata.

Each section focuses on a specific aspect of the audio file handling process, breaking down the implementation into manageable modules. This modular approach aids in both understanding and debugging, allowing for easier maintenance and updates.

The chapter includes meticulous details about each file and function, ensuring that the reader can follow along with the code and understand the purpose and function of each part. This thoroughness is crucial for replicating the setup or modifying it for different applications.

By implementing support for various audio sources, the system becomes versatile and robust, capable of managing different types of audio inputs. This flexibility is beneficial for developing more comprehensive audio control applications.

Leveraging the ESP8266 microcontroller, known for its cost-effectiveness and wireless capabilities, makes the project accessible and practical for many users. The microcontroller's features are well-utilized to handle audio processing tasks efficiently.

In summary, the chapter provides a foundation for understanding and implementing audio control systems using the ESP8266 microcontroller. It offers a detailed and structured approach to software development, ensuring that the reader can effectively

manage and control various audio codecs through a well-organized and modular coding framework.

CONCLUSIONS

The first chapter is devoted to important points to clarify the significance and potential benefits of developing multimedia support system software for ATMega328-based microcontroller systems.

The second chapter is devoted to a comprehensive overview of the components, software, and technical tools necessary for controlling sound codecs using the ESP8266 microcontroller. The chapter began with an introduction to the ESP8266Audio library, highlighting its role and significance in managing audio operations on the ESP8266 platform.

The third chapter is devoted to the implementation of software and technical tools for controlling sound codecs based on the ESP8266 microcontroller provides a comprehensive and detailed guide to the development process. This section highlights the practical steps taken to create and manage various audio file sources through the implementation of specific header files and associated C++ files.

REFERENCES

1. Fu S., Bhavsar P. C. Robotic Arm Control Based on Internet of Things. *IEEE Long Island Systems Applications and Technology Conf.* 2019. Vol. 1. Pp. 1-6.

2. Rao B. N., Sudheer R. Energy Monitoring using IOT. Int. Conf. on Inventive Computation Technologies. 2020. Pp. 868-872.

3. Mesquita J., Guimarães D., Pereira C., Santos F., Almeida L. Assessing the ESP8266 WiFi module for the Internet of Things. *IEEE Int. Conf. on Emerging Technologies and Factory Automation*. 2018. Pp. 784-791.

4. Lekhaa T. R., Rajeshwari S., Sequeira J. A., Akshayaa S. Intelligent Shopping Cart Using Bolt Esp8266 Based on Internet of Things. *Int. Conf. on Advanced Computing Communication Systems*. 2019. Pp. 758-761.

5. de A. Lima C. M., da Silva E. A., Velloso P. B. Performance Evaluation of 802.11 IoT Devices for Data Collection in the Forest with Drones. *IEEE Global Communications Conf*. 2018. Pp. 1-7.

6. Maier A., Sharp Y., Vagapov Y. Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things. *Internet Technologies and Applications*. 2017.

7. Kodali R. K., Soratkal S. MQTT based home automation system using ESP8266. *IEEE Region 10 Humanitarian Technology Conf*. 2016. Pp. 1-5.

8. Yadav R. K., Vohra H. Design architecture and comparison of interactive smart button using HC-05 and ESP8266. *Int. Conf. on Computing Communication and Automation*. 2017. Pp. 982-985.

9. Ertam F., Kilincer I. F., Yaman O., Sengur A. A New IoT Application for Dynamic WiFi based Wireless Sensor Network. *Int. Conf. on Electrical Engineering*. 2020. Pp. 1-4.

10. Mosashvili, Oniani S. Distance Monitoring System with ESP8266 for Industrial Automation Machines. *IEEE Int. Symposium on Smart and Wireless Systems within the Conf. on Intelligent Data Acquisition and Advanced Computing Systems.* 2020. Pp. 1-5.

11. Starodubtsev, Gajniyarov I., Samedov R., Sibogatova A., Antipina I., Zolotareva Y. Animatronic Hand Model on the Basis of ESP8266. *Int. Multi-Conf on Engineering Computer and Information Sciences*. 2019. Pp. 500-503.

12. Khanchuea, Siripokarpirom R. A Multi-Protocol IoT Gateway and WiFi/BLE Sensor Nodes for Smart Home and Building Automation: Design and Implementation. *Int. Conf. of Information and Communication Technology for Embedded Systems*. 2019. Pp. 1-6.

13. Kodali R. K., Mahesh K. S. Low cost ambient monitoring using ESP8266. *Int. Conf. on Contemporary Computing and Informatics*. 2016. Pp. 779-782.

14. Rizal Isnanto R., Eko Windarto Y., Imago Dei Gloriawan J., Noerdiyan Cesara F. Design of a Robot to Control Agricultural Soil Conditions using ESP-NOW Protocol. *Int. Conf. on Informatics and Computing*. 2020. Pp. 1-6.

15. Pertab Rai, Rehman M. ESP32 Based Smart Surveillance System. *Int. Conf. on Computing Mathematics and Engineering Technologies*. 2019.

16. Minchev D., Dimitrov A. Home automation system based on ESP8266. *Int. Symposium on Electrical Apparatus and Technologies*. 2018.

17. Mahmood, Exel R., Bigler T. On clock synchronization over wireless LAN using timing advertisement mechanism and TSF timers. *IEEE Int. Symposium on Precision Clock Synchronization for Measurement Control and Communication*. 2014. Pp. 42-46.

18. Mahmood, Exel R., Sauter T. A simulation-based comparison of IEEE 802.11s timing advertisement and SyncTSF for clock synchronization. *IEEE WorId Conf. on Factory Communication Systems*. 2015. Pp. 1-4.

20. Hoang T. N., Van S., Nguyen B. D. ESP-NOW Based Decentralized Low Cost Voice Communication Systems For Buildings. *Int. Symposium on Electrical and Electronics Engineering*. 2019. Pp. 108-112.

21. ESP-NOW/ URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_now.html (Accessed: 5.04.20240)..

22. ESP-NOW - ESP-IDF Programming Guide v4.0-dev-1191. 2019. Pp. 2-4.

23. Kumar S., et al. Internet of Things is a revolutionary approach for future technology enhancement: a review. Journal of Big Data. 2019. Vol. 6. Pp. 1–21.

24. Noura M., Atiquzzaman M., Gaedke M. Interoperability in Internet of Things: *Taxonomies and Open Challenges. Mobile Network Application*. 2019. Vol. 24, No. 3. Pp. 796–809.

25. IoT application areas. https://iot-analytics.com/top-10-iot-project-application-areas-q3-2016/ Accessed 8 Apr 2020.

26. Wadhwani S., Singh U., Singh P., Dwivedi S. Smart Home Automation and Security System using Arduino and IOT. *Int. Research Journal of Engineering and Technology (IRJET)*. 2018. Vol. 5, No. 2. Pp. 1357–1359.

27. Pasha S. Thingspeak Based Sensing and Monitoring System for IoT with Matlab Analysis. *Int. Journal of New Technology and Research (IJNTR)*. 2016. Vol. 2, No. 6. Pp. 19-23.

28. Kiliç T., Bayir E. An Investigation on Internet of Things Technology (IoT) In Smart Houses. *Int. Journal of Engineering Research and Development*. 2017. Vol. 9. Pp. 197–206.

29. Çeltek S. A., Soy H. An application of building automation system based on wireless sensor/actuator networks. *Int. Conf. on Application of Information and Communication Technologies (AICT)*. 2015. Pp. 450-453.

30. Wahab N. A., et al. Performance of Environmental and Energy Audit for Manufacturing Industrial Buildings. *Indonesian Journal of Electrical Eng. and Computer Science (IJEECS)*. 2018. Vol. 12, No. 2. Pp. 534-541.

31. Chaudhury S., Paul D., Mukherjee R., Haldar S. Internet of Thing based healthcare monitoring system. 2017.

32. 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON). Bangkok. 2017. Pp. 346-349.

33. Visconti P., et al. IoT-oriented software platform applied to sensors-based farming facility with smartphone farmer app. *Bulletin of Electrical Engineering and Informatics (BEEI)*. 2020. Vol. 9, No. 3. Pp. 1095-1105.

34. Salih N. A. J., Hasan I. J., Abdulkhaleq N. I. Design and implementation of a smart monitoring system for water quality of fish farms. *Indonesian Journal of Electrical Engineering and Computer Science (IJEECS)*. 2019. Vol. 14, No. 1. Pp. 44-50.

35. Van De Moosdijk J., Visser D. Car security: remote keyless entry and go. Revision 232. June 2009.

36. Mason S. Vehicle remote keyless entry systems and engine immobilisers: Do not believe the insurer that this technology is perfect. *Computer Law & Security Review*. 2012. Vol. 28. Pp. 195-200.

37. Agarwal R., Boominathan P. Vehicle Security System Using IoT Application. *Int. Research Journal of Engineering and Technology (IRJET)*. 2018. Vol. 5, No. 4. Pp. 910-912.

38. Datasheet Espressif Systems. Espressif Smart Connectivity Platform: ESP8266. 2013. URL: https://www.electroschematics.com/wp-content/uploads/2015/02/esp8266-datasheet.pdf (Accessed: 5.04.20240)..

39. Datasheet ESP 12E PIN OUT. URL: https://components101.com/wireless/esp12e-pinout-datasheet; https://www.make-it.ca/nodemcu-arduino/nodemcu-details-specifications/.

40. Arduino software. URL: https://www.arduino.cc/.

41. Arduino IDE online version. URL: https://www.arduino.cc/en/Main/Software (Accessed: 5.04.20240)..

42. Arduino firebase Master. URL: https://github.com/FirebaseExtended/firebase-arduino (Accessed: 5.04.20240)..

43. ArduninoJson. URL: https://github.com/bblanchon/ArduinoJson/tree/5.x (Accessed: 5.04.20240)..

44. DHT by adafruit. URL: https://github.com/adafruit/DHT-sensor-library.

45. Adafruit universal sensor. URL: https://github.com/adafruit/Adafruit_Sensor (Accessed: 5.04.20240)..

46. ESP8266 wi-fi module. URL: https://github.com/esp8266/Arduino (Accessed: 5.04.20240)..

47. Alsalemi A., et al. Real-Time Communication Network Using Firebase Cloud IoT Platform for ECMO Simulation. *IEEE Int. Conf. on Internet of Things (iThings), Green Computing and Communications (GreenCom), Cyber, Physical and Social Computing (CPSCom), Smart Data (SmartData)*. 2017. Pp. 178-182.

48. This system- IoT and Cloud based system Database Data Retreived as Json Document. URL: https://nubarealtimedb.firebaseio.com/.json.

49. Firebase Rest API Usage. URL: https://firebase.google.com/docs/reference/rest/database#section-cond-ifmatch.

50. MIT Tool. URL: http://appinventor.mit.edu/explore/.

51. MIT project space for the app development of this IoT and Cloud based system. URL: http://ai2.appinventor.mit.edu/#5948786283777395 2 (Accessed: 5.04.20240).

Annex A

The flowchart of the algorithms

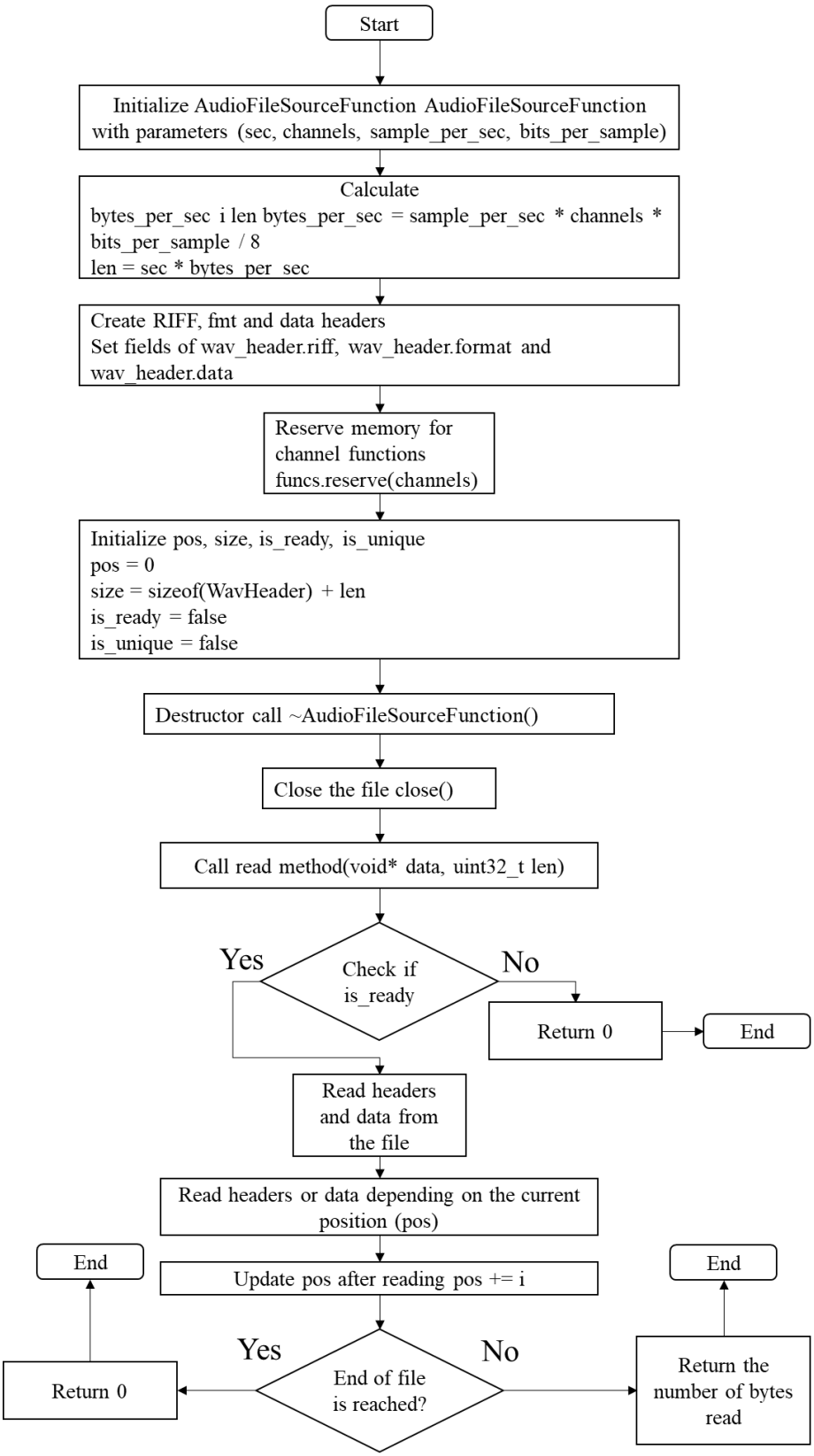


Figure A.1 - AudioFileSourceFunction.cpp flowchart of the algorithms

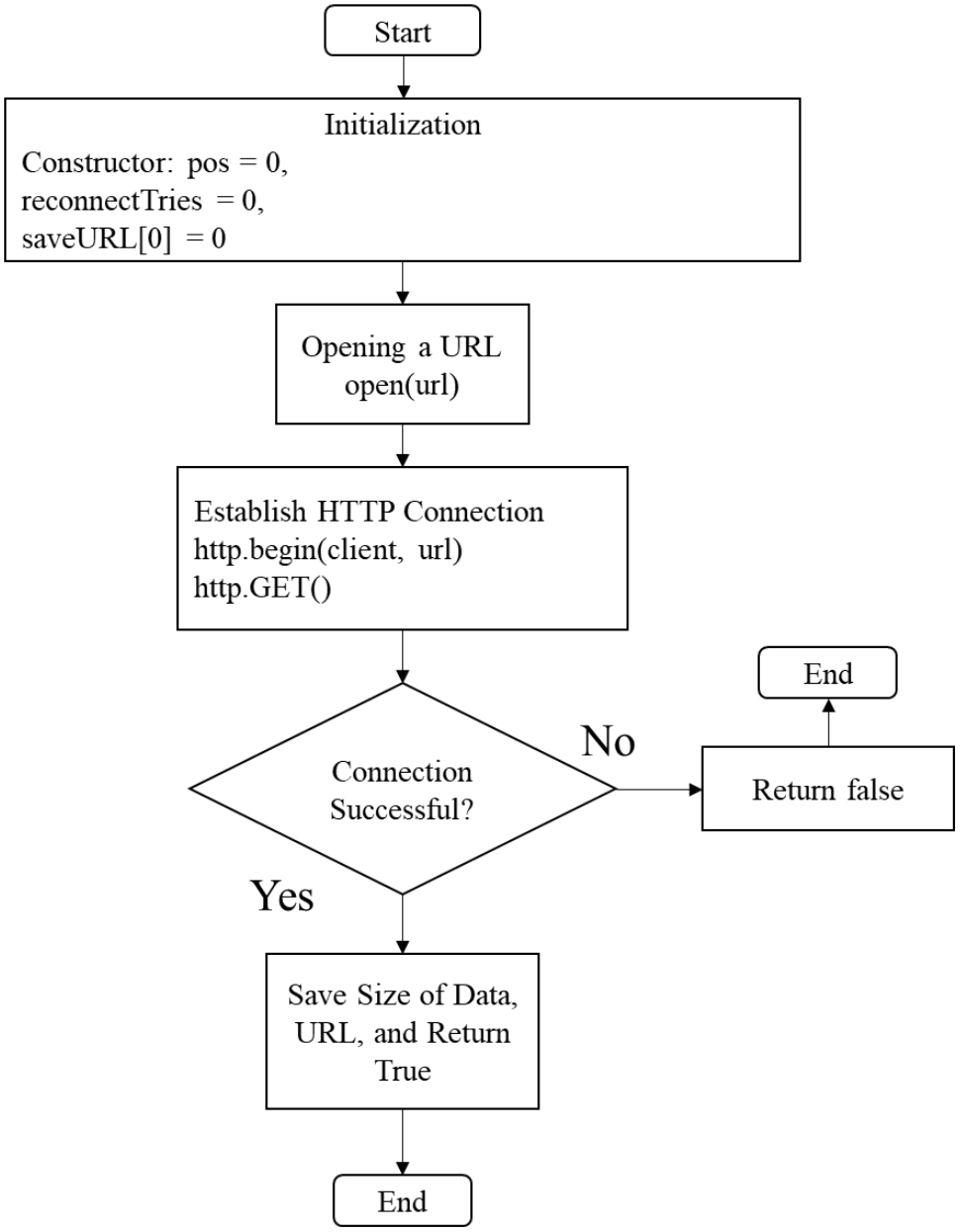


Figure A.2 - AudioFileSourceHTTPStream.cpp flowchart of the algorithms

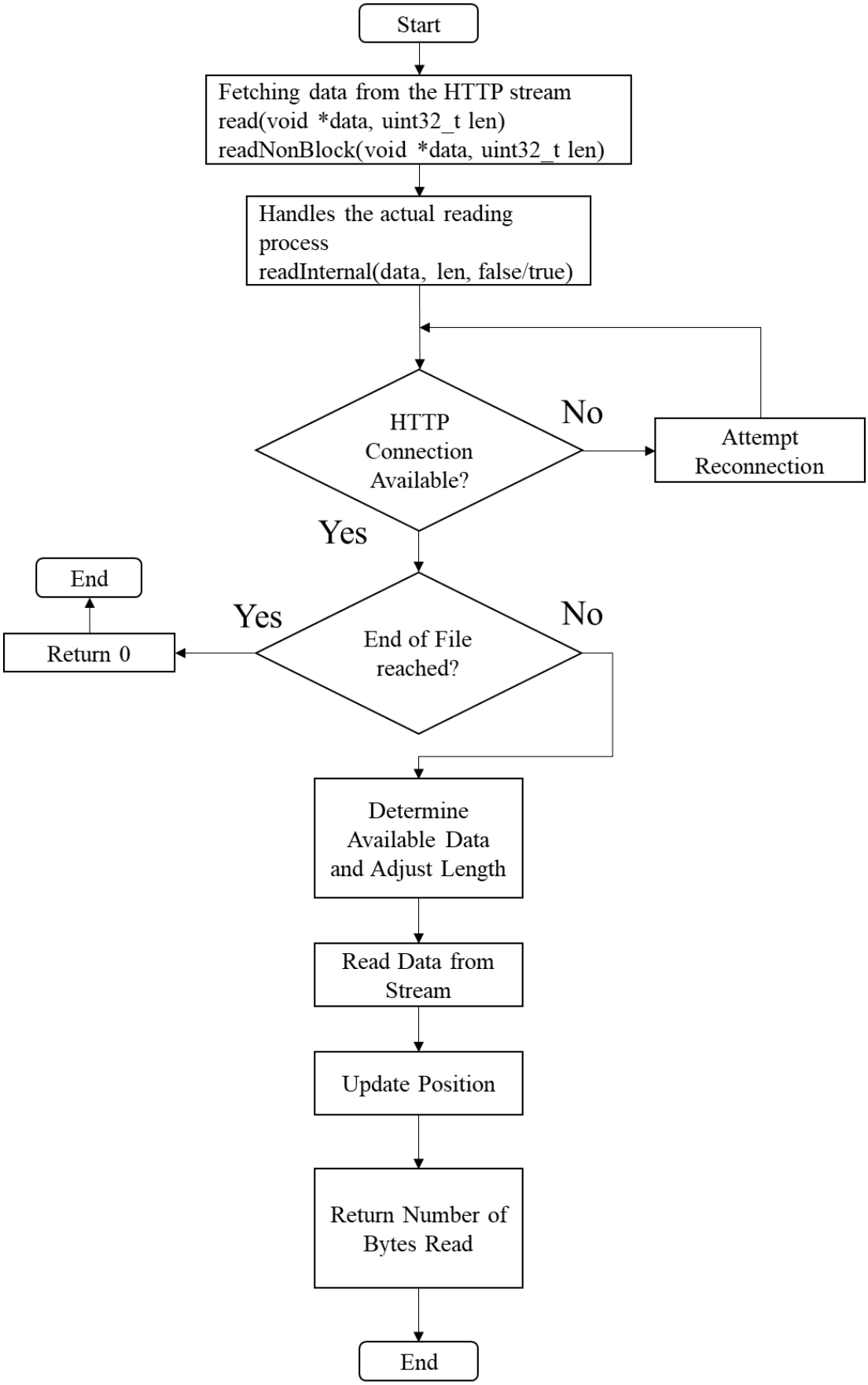


Figure A.3 - AudioFileSourceHTTPStream.cpp flowchart of the algorithms

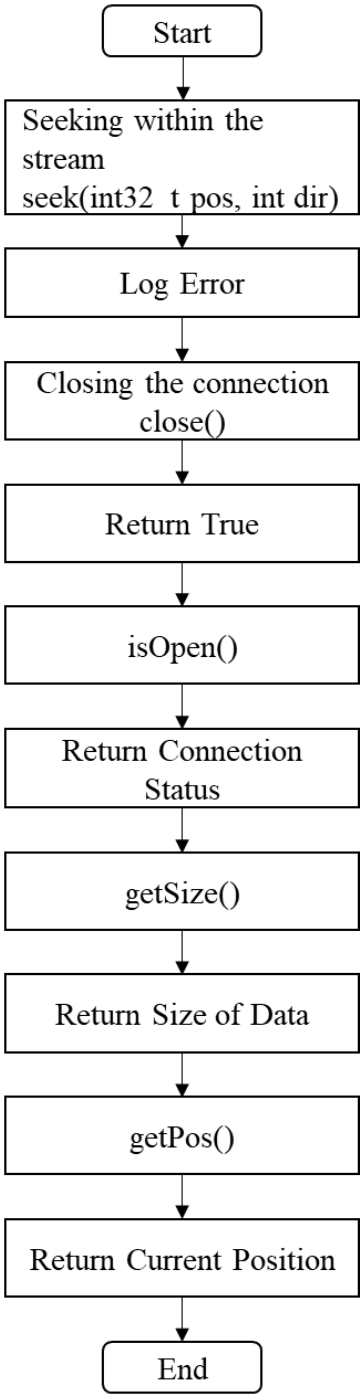
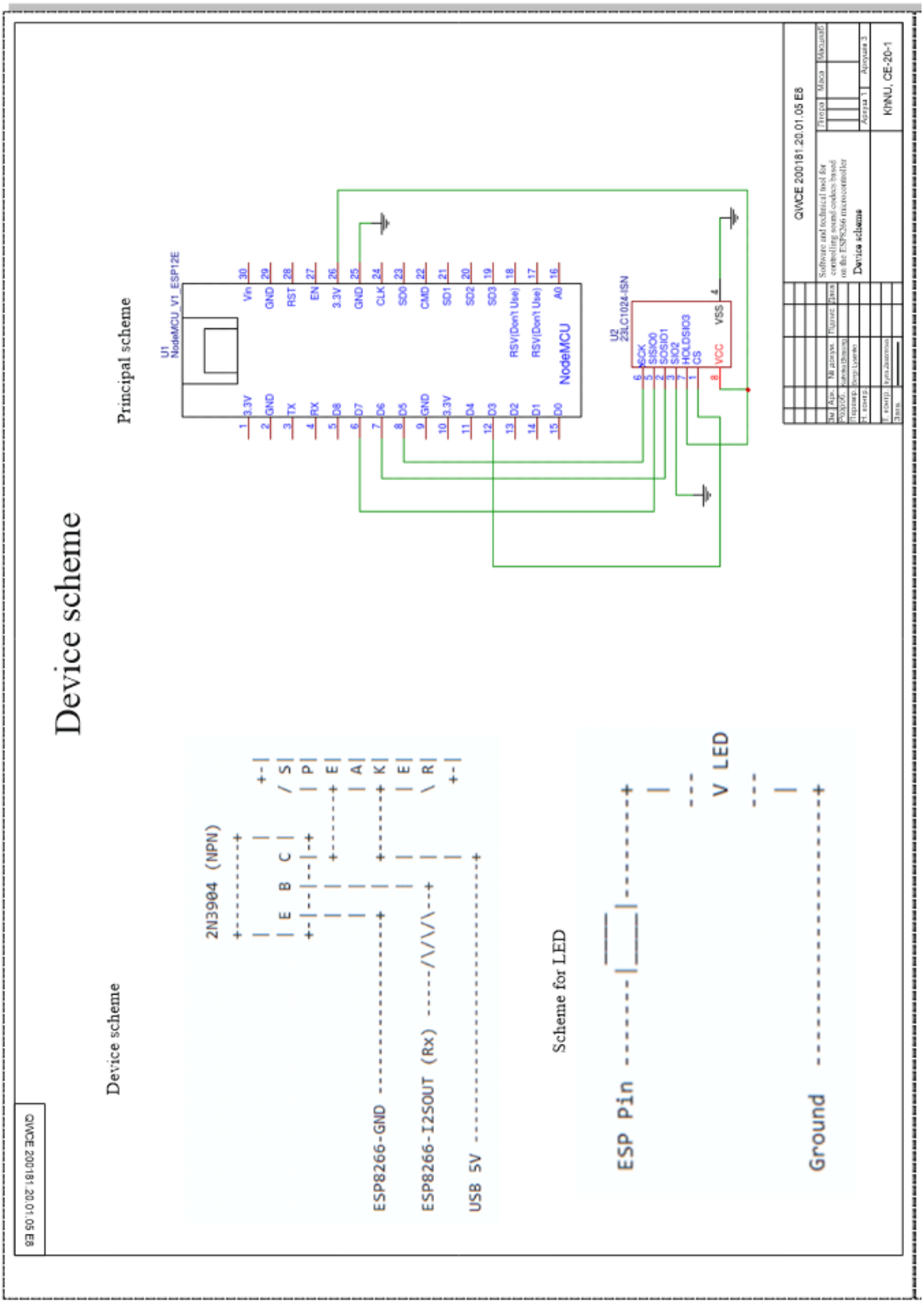


Figure A.4 - AudioFileSourceHTTPStream.cpp flowchart of the algorithms

Annex B

(required)

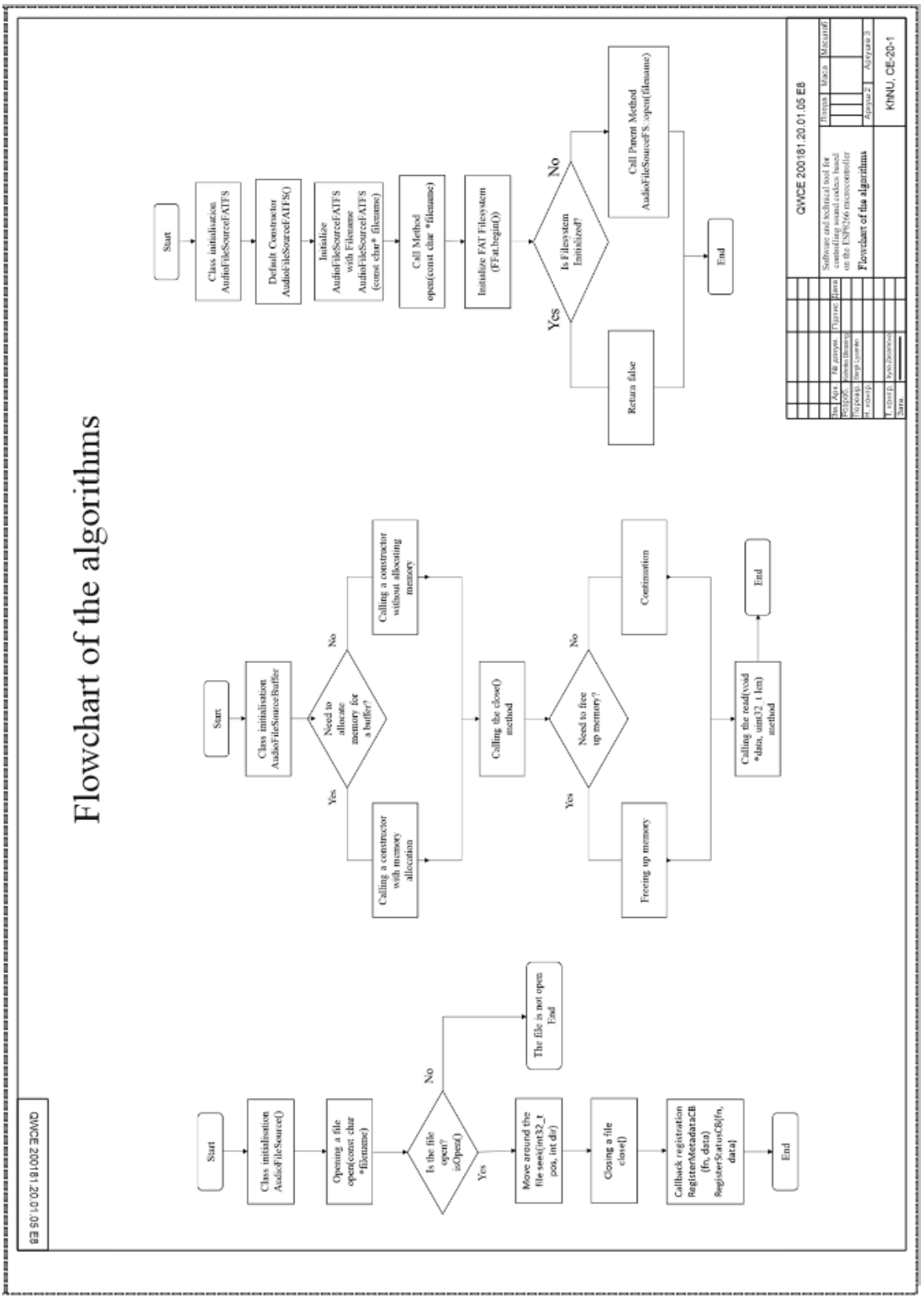
Copy of the drawing “Device scheme”



Annex C

(required)

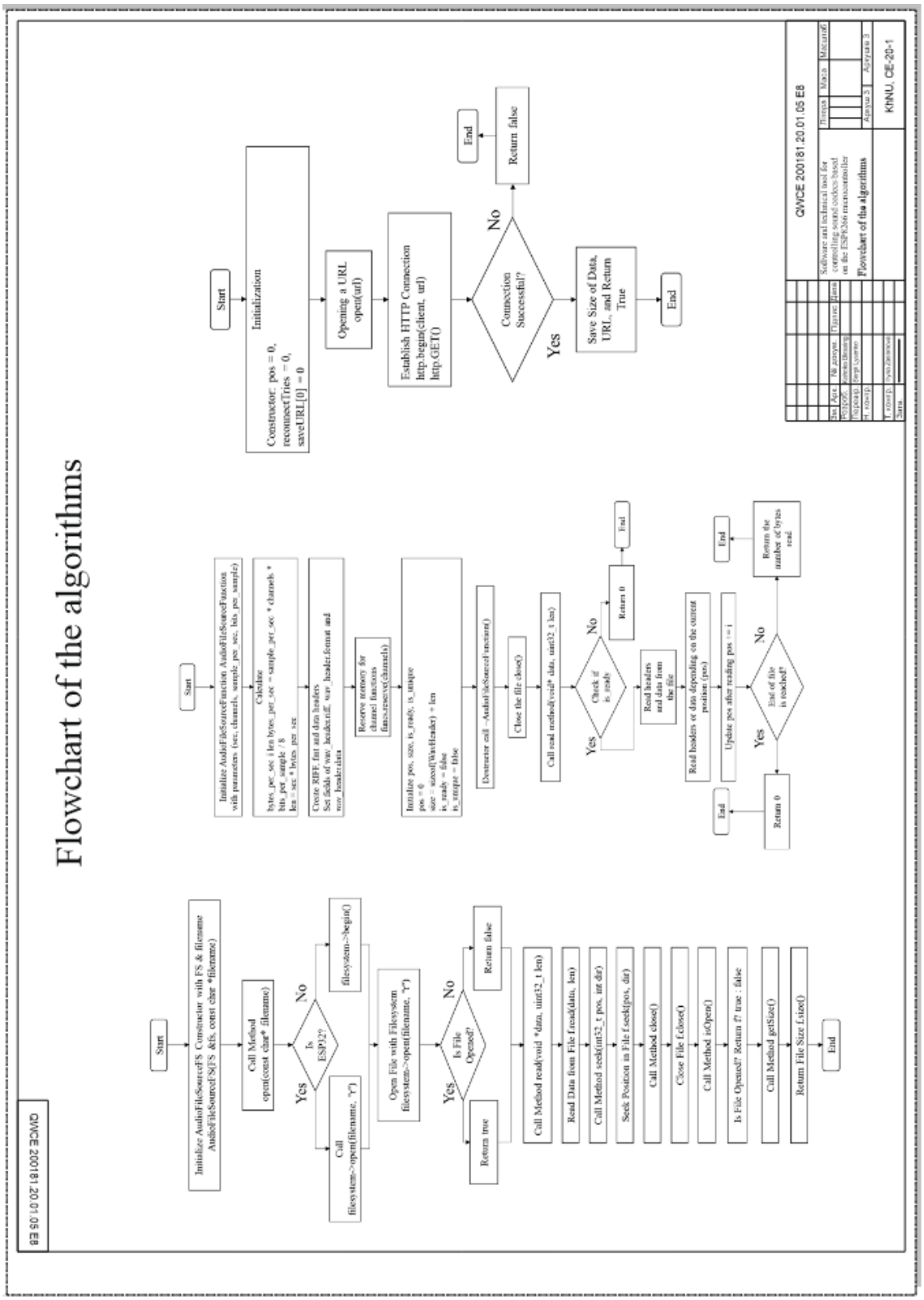
Copy of the drawing “Flowchart of the algorithms”



Annex D

(required)

Copy of the drawing «Flowchart of the algorithms»



UNICHECK
by Turnitin

Ім'я користувача:
Кафедра КІ

ID перевірки:
1016350579

Дата перевірки:
12.06.2024 07:36:55 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
12.06.2024 10:50:50 EEST

ID користувача:
100005591

Назва документа: Блессінг_Програмно-технічний засіб керування звуковими кодеками на основі мікроконтр...

Кількість сторінок: 77 **Кількість слів:** 14611 **Кількість символів:** 117783 **Розмір файлу:** 311.58 KB **ID файлу:** 1016154246

2.15%
Схожість

Найбільша схожість: 1.04% з джерелом з Бібліотеки (ID файлу: 1016154247)

1.54% Джерела з Інтернету	47	Сторінка 79
1.87% Джерела з Бібліотеки	82	Сторінка 79

0.37% Цитат

Цитати	5	Сторінка 80
Посилання	1	Сторінка 80

0%
Вилучень

Немає вилучених джерел

UNICHECK
by Turnitin

Ім'я користувача:
Кафедра КІ

ID перевірки:
1016350580

Дата перевірки:
12.06.2024 07:46:48 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
12.06.2024 10:51:23 EEST

ID користувача:
100005591

Назва документа: Blessing_Software and technical tool for controlling sound codecs based on the ESP8266 microcontroller

Кількість сторінок: 76 **Кількість слів:** 16682 **Кількість символів:** 116339 **Розмір файлу:** 330.19 KB **ID файлу:** 1016154247

5.14%
Схожість

Найбільша схожість: 3.34% з Інтернет-джерелом (https://github.com/tuanpmt/esp-audio-player)

4.68% Джерела з Інтернету	124	Сторінка 78
1.08% Джерела з Бібліотеки	4	Сторінка 79

2.65% Цитат

Цитати	21	Сторінка 80
Посилання	1	Сторінка 81

0%
Вилучень

Немає вилучених джерел

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 28%**

	Документ		Сумарний збіг по Базі Даних	
ID: 129814 Назва: БКР Software and technical tool for controlling sound codecs based on the ESP8266 microcontroller Додано в БД: 2024-06-12 Автора: Kahoka Blessing Керівники: S.M. Lysenko Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	98184	789	236 (0%)	3 (0%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 7.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 15%**

	Документ		Сумарний збіг по Базі Даних	
ID: 129815 Назва: БКР Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266 Додано в БД: 2024-06-12 Автора: Кахока Блесінг Керівники: С.М. Лисенко Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	100270	818	7260 (7%)	75 (9%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Кахока Блессінг

Тема: Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___62___

1. Короткий зміст роботи та прийнятих рішень: Метою дипломної роботи є розроблення програмно-технічного засобу керування звуковими кодеками на основі мікроконтролера ESP8266.

2. Висновок про відповідність роботи дипломному завданню: Робота *повністю* відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: Перший розділ присвячено важливим моментам, що пояснюють значення та потенційні переваги розробки програмного забезпечення системи підтримки мультимедіа для мікроконтролерних систем на базі ATMega328.

Другий розділ присвячено всебічному огляду компонентів, програмного забезпечення та технічних засобів, необхідних для керування звуковими кодеками за допомогою мікроконтролера ESP8266. Розділ починається зі знайомства з бібліотекою ESP8266Audio, висвітлення її ролі та значення в управлінні аудіоопераціями на платформі ESP8266.

Третій розділ присвячений реалізації програмних і технічних засобів для управління звуковими кодеками на базі мікроконтролера ESP8266 і містить вичерпний і детальний посібник з процесу розробки. У цьому розділі висвітлено практичні кроки, зроблені для створення та управління різними джерелами аудіофайлів за допомогою реалізації спеціальних заголовних файлів і пов'язаних з ними C++-файлів.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: недостатня увага аналогічним технічним рішенням.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному технічному рівні.

8. Інші зауваження: _____

9. Оцінка дипломної роботи: добре, С

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) _____

Корецька Л.О., к.т.н., доцент, доцент кафедри Автоматизації, комп'ютерно-інтегрованії технологій та робототехніки

"11" 06 2024 р. _____(підпис)

Завідувачу кафедри КІІС
д-р.техн.наук, проф. Говорущенко Т. О.

Кахока Блессінг

ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2ін-20-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

22 квітня 2024 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Програмно-технічний засіб керування звуковими кодеками на основі мікроконтролера ESP8266

Автор: Кахока Блессінг

Спеціальність: 123– Комп'ютерна інженерія

Освітня програма: освітньо-професійна

Науковий керівник: Лисенко Сергій Миколайович, д.т.н, професор

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) запозичення розміщені в розділах аналізу існуючих аналогів та прототипів, які не описують безпосередньо авторське дослідження і не стосуються результатів роботи;
2) усі запозичення фрагментарні, або мають належним чином оформленні посилання;
3) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 10-40 джерелами на один фрагмент речення;

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості, складає 5.14% і адресується до 124 першоджерела, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ С. М. Лисенко

Гарант ОП _____ С.М. Лисенко

Завідувач кафедри КІІС _____ Т. О. Говорущенко