**Cyfrowy Equalizer**

**Cel projektu**

Celem projektu jest zaprojektowanie i oprogramowanie odtwarzacza muzycznego za pomocą mikrokontrolera. Kluczową funkcjonalnością układu ma być equalizer pozwalający na filtrowanie i zmienianie barwy odtwarzanego dźwięku. Mikrokontroler będzie odtwarzał utwory zapisane na karcie SD i umożliwiał zdalną kontrolę poprzez moduł bluetooth.

**Obsługiwane funkcjonalności**

* Odczyt plików wave audio z karty microSD
* Odtwarzanie plików dźwiękowych
* Korektor (equalizer) pozwalający zmieniać tony dżwięku
* Odtwarzacz umożliwiający wybór utworów, przewijanie, pauzowanie i konfigurowanie equalizera
* Interfejs zdalnej obsługi bluetooth pozwalający na kontrolowanie odtwarzacza
* Aplikacja do zdalnej obsługi kontrolera na urządzenia z systemem Linux

**Hardware**

* Mikrokontroler STM32F407G-DISC1

# Moduł czytnika kart microSD

# Moduł Bluetooth HC-06 ZS-040

# Karta pamięci Goodram microSD 16GB

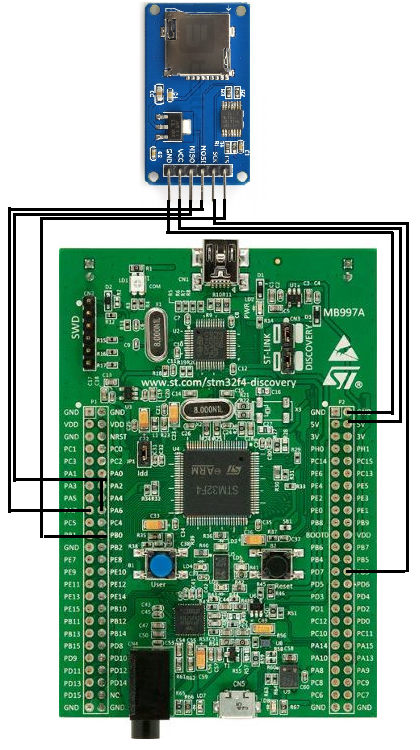
**Realizacja projektu**

1. **Zapoznanie się ze specyfikacją mikrokontrolera i modułów, oraz złożenie prototypu**

Do połączenia komponentów z kontrolerem wykorzystano kable stykowe żeńsko-żeńskie.

**Karta SD**

Moduł czytnika kart microSD podłączono do pinów PA5, PA6, PB3 oraz PB0, do napięcia 5V. Komunikacją z kartą SD odbywa się z wykorzystaniem protokołu SPI.



**Moduł Bluetooth**

Moduł kompunikacji Bluetooth ZS-040 podłączono do pinów PA3 i PA4 oraz do napięcia 5V. Komunikacją z modułem odbywa się za pomocą protokołu UART przy prędkości 9600 baudów.

1. **Przygotowanie środowiska do pracy z mikrokontrolerem w systemie Ubuntu**

Projekt realizowany jest w języku C++ w standardzie C++17 z wykorzystaniem bibliotek HAL oraz FatFs. Za kompilację odpowiada CMake wykorzystujący kompilator gcc-arm-none-eabi. Do programowania pamięci Flash mikrokontrolera używany jest program OpenOCD.

Jako zintegrowane środowisko programistyczne używany jest program CLion.

1. **Implementacja inicjalizacji mikrokontrolera i jego poszczególnych modułów**

Do częściowej generacji kodu konfiguracji zegara oraz kodu inicjalizacji poszczególnych komponentów i bibliotek wykorzystywany jest program CubeMX.

Inicjalizacja komponentów użuwanych przez program kontrolera odbywa się w klasie **Application**. Odpowiada ona za inicjalizację:

* Biblioteki HAL
* Ustawień zegara
* Pinów do obsługi diod LED
* Komunikacji SPI z modułem kart microSD
* Systemu plików FAT dla karty microSD

Obsługuje ona również przerwania zegarowe.

1. **Implementacja prostego odtwarzania dźwięku wave audio z karty microSD**

Za odczyt metadanych audio oraz próbek dźwięku odpowiada klasa **WavAudioReader**. Umożliwia ona ciągły odczyt sekwencyjny danych, a także wykonuje walidację pliku, wykrywając błędy i informując o nieobsługiwanych formatach i parametrach.

Klasa **AudioPlayer** odtwarza pliki audio wykorzystując klasę **WavAudioReader**. Odtwarzanie porzebiega z wykorzystaniem sterowników BSP płytki discovery.

1. **Implementacja graficznego equalizera z wykorzystaniem szybkiej transformacji Fouriera**

Klasa **AudioPlayer** umożliwia filtrowanie dźwięku przed jego odtwarzaniem. Interfejs filtrowania dostarcza klasa **AudioFilter**, która wykonuje skalowanie próbek z wartości całkowitoliczbowych do wartości zmiennoprzecinkowych (-1, 1), rozdzielając ciąg próbek na poszczególne kanały, a następnie dla próbek każdego kanału wywołuje metodę **processNormalizedSamples**, która przetwarza znormalizowane próbki. Następnie przefiltrowane wartości rzeczywiste są konwertowane z powrotem do wartości całkowitoliczbowych.

Klasa **FFTAudioFilter** implementuję metodę **processNormalizedSamples**, wykonując Szybką Transformację Fouriera. Transformacja wykonywana jest bez dodatkowej pamięci, z wersją ze zwiniętą rekurencją w iterację. Algorytm wykorzystuje decymację częstotliwościową (Decimation-in-Frequency). Klasa wywołuję następnie metodę **processFFT** i wykonuje odwrotną transformację Fouriera.

Klasa **DigitalEqualizerFilter** implementuje metodę **processFFT**, w której wykonuje filtrowanie próbek transformaty. Dostępne parametry konfiguracji to podbicie wartości amplitudy dla wszystkich częstotliwości oraz podbicie amplitudy w bandach oktawy ISO (zestaw dziesięciu bandów).

1. **Przygotowanie funkcjonalnego interfejsu odtwarzacza.**

Klasa **Playlist** udostępnia listę plików audio do odtwarzania z danego katalogu karty SD do odtwarzania przez klasę **PlaylistPlayer**. PlaylistPlayer wykorzystuje klasę **AudioPlayer** wrapując jego funkcjonalności oraz udostępniając dodatkową funkcjonalność odtwarzania według playlisty. Lista funkcjonalności:

* wybór utworu z playlisty
* zmiana poziomu głośności
* zmiana pozycji odtwarzania
* pauza, stop
* ustawienie filtru audio (equalizera graficznego)

1. **Publikacja interfejsu odtwarzacza poprzez moduł bluetooth.**

Za komunikację bluetooth odpowiada klasa **BluetoothCommunicationProvider** umożliwiająca wysyłanie i odbieranie wiadomości. Inicjalizuje ona moduł bluetooth i implementuje prosty protoków komunikacji przesyłając wiadomości o określonym rozmiarze.

Wiadomości bluetooth zdefiniowane zarówno dla mikrokontrolera i aplikacji zdalne są zdefiniowane w bibliotece **player-protocol** automatyzującej proces serializacji dancych oraz definiującej interfejs odtwarzacza. Obsługuje trzy typy wiadomości:

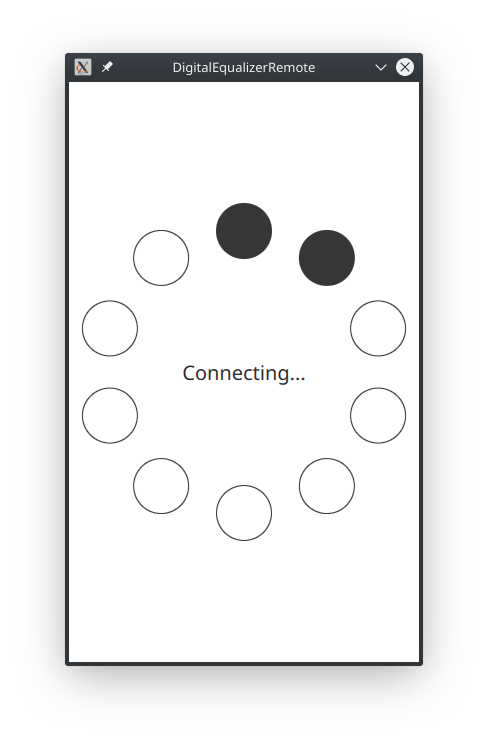
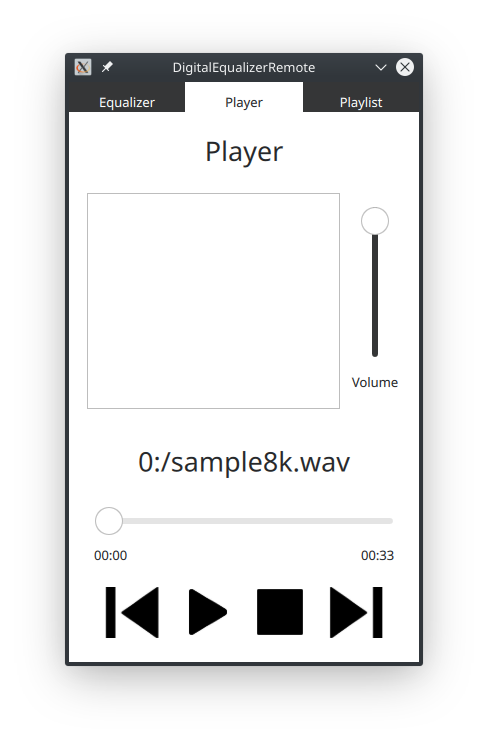
* changed – periodyczne wiadomości wysyłane przez serwer (kontroler)
* request – wiadomości wysyłane przez klienta (aplikację)
* response – odpowiedzi serwera (kontrolera)

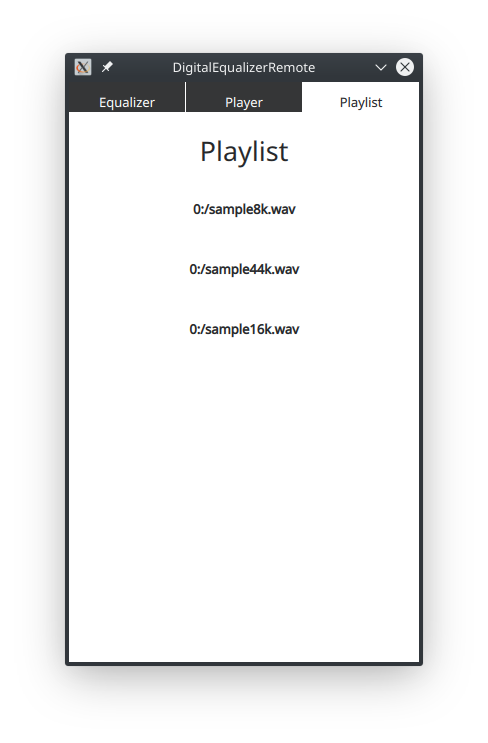
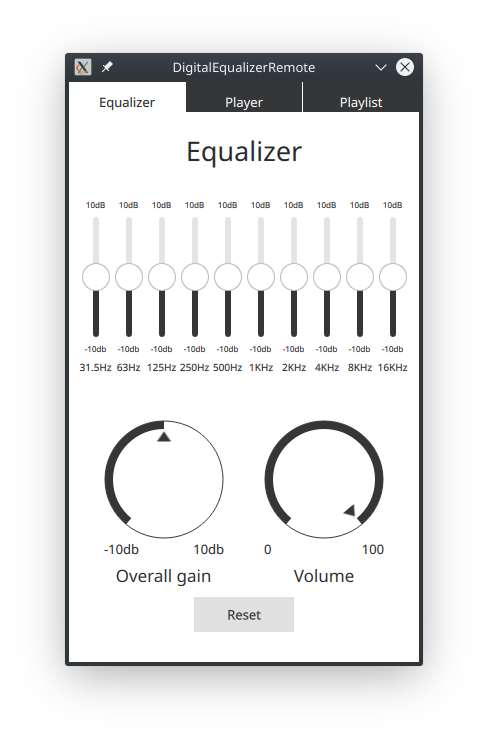
Po stronie mikrokontrolera za obsługę requestów i wysyłanie wiadomości periodycznych odpowiadają klasy **RequestHandler** oraz **NotificationService**.

1. **Przygotowanie aplikacji obsługującej mikrokontrolerowy odtwarzacz przez bluetooth.**

Aplikacja klienta odtwarzacza jest programem napisanym w C++17 z wykorzystaniem biblioteki Qt5. Implementuje on wzorzec strukturalny MVC z widokiem napisanym w języku QML oraz modelem i kontrolerem zaimplementowanymi po stronie C++.

Do kompunikacji użyto modułu Bluetooth biblioteki Qt, wykorzystując podobnie jak po stronie serwera bibliotekę **player-protocol**.





Klasy modelowe:

* **ConnectionModel** – użyty podczas łączenia się z mikrokontrolerem
* **PlayState** – status odtwarzania
* **EqualizerModel** – konfiguracja parametrów equalizera
* **PlayerModel** – model odtwarzacza zawierający cały jego stan

Klasy kontrolera:

* **PlayerController** – odbiera interakcje użytkownika, przekazując je dalej do klas modelu i komunikacji z serwerem.

Widok:

Dostępne są trzy widoki dla użytkownika

* Equalizer – umożliwia zmienianie parametrów equalizera oraz głośności odtwarzania
* Player – domyślny widok, pozwalający na kontrolowanie obecnie odtwarzanego utworu
* Playlist – pozwala na wybór utworu do odtwarzania z playlisty

Podczas nawiązywania połączenia wyświetlany jest użytkownikowi indykator, a w razie błędu dialog z pytaniem o ponowienie połączenia.

1. **Napotkanie trudności implementacyjne**

Podstawowym problemem implementacji była wydajność mikroprocesora. W finalnej wersji pliki audio z częśtotliwością 44KHz nie są odtwarzane z pełną prędkością. Pliki o niższych częstotliwościach odtwarzają się poprawnie. Przyczyną niskiej wydajności są jest między innymi komunikacja z kartą SD – odczyt danych zajmował największą część przetworzenia bufora przed przekazaniem go do odtwarzacza.

Aby zwiększyć wydajność filtrowania próbek audio konieczne było zwiększenie bufora odczytu i filtrowanie go w częściach, co umożliwiło płynne odtwarzanie plików audio.

Problemem, który utrudniał implementację były również obserwowalne modele. Po stronie kontrolera zmiana stanu odtwarzacza powodowała wysłanie wiadomości do aplikacji. Po stronie aplikacji zmiana w modelu propagowała się do widoku, a następnie do kontrolera. W przypadku kiedy zmiany zostały obsłużone niepoprawnie mogło dojść do nieskończonej pętli komunikacji między kontrolerem a aplikacją.

Przykładem kiedy zatrzymanie daleszego propagowania wiadomości było trudne jest np. konfiguracja equalizera. Po stronie aplikacji zapisuje ona typy double natomiast po stronie mikrokontrolera typy float. Konieczne było więc wykonanie porównania z dodatkowym epsilonem, aby zatrzymać dalsze przekazywanie informacji o zmianie wartości.

Trudnością w implementacji po stronie kontrolera była również wydajność programu w tybie Debug. Tak skompilowany program działał wolniej i nie odtwarzał utworów w czasie rzeczywistym, co utrudniło proces debugowania.

**Repozytoria**

Kod źródłowy aplikacji kontrolującej odtwarzacz

https://github.com/artudi54/DigitalEqualizerRemote

Kod źródłowy programu mikrokontrolera

https://github.com/artudi54/DigitalEqualizer

Biblioteka komunikatów odtwarzacza klient-serwer

https://github.com/artudi54/player-protocol

**Źródła**

Strona producenta z dokumentacją mikrokontrolera:

<https://www.st.com/en/microcontrollers-microprocessors/stm32f407vg.html>

Szablonowy projekt CMake dla płytek STM32F4xx

<https://github.com/ahessling/STM32F4Template>

Specyfikacja kontenera WAVE

http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

Przykładowy kod equalizera plików WAVE napisany w C

<https://github.com/kappalabs/fft_band_equalizer>

Przykładowy kod equalizera napisany w C++ wykorzystujący biblioteki do FFT I filtrowania dźwięku

https://sourceforge.net/projects/equalizerapo/

Artykuł o wykorzystaniu szybkiej transformacji Furiera w equalizerach.

[http://www.zytrax.com/tech/audio/equalization.html#overview](http://www.zytrax.com/tech/audio/equalization.html" \l "overview)

Decymacja częstotliwości

<https://www.brainkart.com/article/Decimation-In-Frequency-(DIFFFT)_13033/>