


Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Systemów Elektronicznych

**Szymon Buś**

Numer indeksu: 246136

Praca Inżynierska

# Realizacja gitarowych efektów dźwiękowych z wykorzystaniem metod cyfrowego przetwarzania sygnałów

Praca wykonana pod kierunkiem:
dr hab. inż. Konrad Jędrzejewski

Warszawa, 2015

**Realizacja gitarowych efektów dźwiękowych z wykorzystaniem metod cyfrowego przetwarzania sygnałów**

W pracy przedstawiono realizację cyfrowego dźwiękowego multiefektu gitarowego. Prace były wykonywane w dwóch fazach. W pierwszej z nich powstała aplikacja w środowisku Matlab z graficznym interfejsem użytkownika, pozwalająca na zastosowanie efektów dźwiękowych do sygnałów zapisanych w postaci plików WAV (ang. Waveform Audio File Format). W drugiej fazie zaprojektowano i zrealizowano system czasu rzeczywistego w oparciu o ewaluacyjny zestaw mikroprocesorowy i kodek audio. System jest przeznaczony do realizacji efektów dźwiękowych na sygnale wyjściowym gitary elektrycznej. W pracy opisano wykorzystane algorytmy, a także sposób ich implementacji.
**Słowa kluczowe:** efekt dźwiękowy, mikroprocesor, cyfrowe przetwarzanie sygnałów.

**Realization of guitar audio effects with use of methods of digital signal processing**

In the thesis, the realization of digital guitar audio multieffect was presented. Work was done in two stages. In the first stage, the application with a graphical user interface was created in Matlab, which enables to use audio effects for signals saved as WAV (Waveform Audio File Format) files. In the second stage, the real-time system based on a microcontroller and audio codec was designed and realized. The system is designed to perform audio effects on the output signal of electric guitar. In the thesis, the algorithms used and the methods of their implementation were described.
**Keywords:** audio effect, microprocessor, digital signal processing.

# Spis treści

**Wprowadzenie** **6**

Wstęp . . . . . 6

Układ pracy . . . . . 6

**1 Omówienie gitarowych efektów dźwiękowych** **7**

1.1 Efekty dźwiękowe . . . . . 7

1.1.1 Efekty gitarowe . . . . . 7

1.1.2 Wykorzystanie metod cyfrowego przetwarzania sygnałów . . . 8

1.2 Delay . . . . . 9

1.2.1 Wersja FIR . . . . . 9

1.2.2 Wersja IIR . . . . . 9

1.3 Reverb . . . . . 10

1.4 Flanger . . . . . 12

1.4.1 Wersja FIR . . . . . 12

1.4.2 Wersja IIR . . . . . 13

1.5 Tremolo . . . . . 13

1.6 Overdrive . . . . . 13

1.6.1 Cyfrowy efekt Overdrive . . . . . 14

1.7 Pitch Shifter . . . . . 15

1.7.1 Algorytm wokodera fazowego . . . . . 16

1.7.1.1 Quasi-stacjonarny model sinusoidalny sygnału . . . . 16

1.7.1.2 Definicje skalowania w czasie i skalowania wysokości 16

1.7.1.3 Analiza . . . . . 18

1.7.1.4 Skalowanie w dziedzinie czasu . . . . . 19

1.7.1.5 Synteza . . . . . 20

1.7.1.6 Podsumowanie algorytmu wokodera fazowego . . . . 20

**2 Symulacja gitarowych efektów dźwiękowych 22**
2.1 Wprowadzenie . . . 22
2.2 Struktura programu . . . 22
2.3 Implementacja algorytmów . . . 23
2.3.1 Delay . . . 24
2.3.2 Reverb . . . 25
2.3.3 Flanger . . . 26
2.3.4 Tremolo . . . 27
2.3.5 Overdrive . . . 27
2.3.6 Pitch Shifter . . . 28
2.4 Obsługa aplikacji przez użytkownika . . . 30

**3 Realizacja efektów gitarowych w czasie rzeczywistym 33**
3.1 Wprowadzenie . . . 33
3.2 Elementy systemu . . . 33
3.2.1 Zestaw ewaluacyjny STM32F4 Discovery . . . 33
3.2.2 Moduł kodeka audio MMCodec01 . . . 34
3.2.3 Połączenie elementów systemu . . . 35
3.3 Oprogramowanie . . . 35
3.3.1 Struktura projektu . . . 36
3.3.2 Konfiguracja kodeka audio . . . 36
3.3.3 Konfiguracja interfejsu $I^2S$ . . . 40
3.3.4 Konfiguracja przerwań . . . 42
3.3.5 Realizacja efektów dźwiękowych . . . 42
3.3.5.1 Delay FIR . . . 44
3.3.5.2 Delay IIR . . . 45
3.3.5.3 Flanger FIR . . . 45
3.3.5.4 Flanger IIR . . . 46
3.3.5.5 Overdrive . . . 47
3.3.5.6 Tremolo . . . 47
3.3.6 Obsługa systemu przez użytkownika . . . 48

**4 Badanie właściwości zrealizowanych efektów gitarowych 49**
4.1 Test działania systemu w warunkach rzeczywistych . . . 49
4.2 Badanie wybranych efektów gitarowych . . . 49

4.2.1 Stanowisko pomiarowe . . . . . 49
4.2.2 Badanie działania systemu przy braku modyfikacji sygnału . . 50
4.2.3 Badanie efektu Overdrive . . . . . 51
4.2.4 Badanie efektu Tremolo . . . . . 52

**5 Podsumowanie** **54**

# Wprowadzenie

## Wstęp

Niniejsza praca inżynierska dotyczy efektów dźwiękowych stosowanych do modyfikacji brzmienia gitary elektrycznej oraz metod cyfrowego przetwarzania sygnałów pozwalających na ich realizację. Celem pracy było stworzenie aplikacji komputerowej pozwalającej na symulację omawianych efektów dźwiękowych poprzez zastosowanie ich do przetwarzania sygnałów zapisanych w plikach audio, a następnie opracowanie i wykonanie systemu czasu rzeczywistego realizującego efekty dźwiękowe na sygnale elektrycznym wytwarzanym przez przetwornik elektromagnetyczny gitary elektrycznej.

## Układ pracy

Praca składa się z pięciu rozdziałów. W pierwszym rozdziale omówiono popularne efekty dźwiękowe stosowane przez gitarzystów. W rozdziale tym zawarte zostały formuły matematyczne opisujące każdy z efektów, algorytmy, które zaimplementowano w ramach niniejszej pracy, informacje o sposobach wykorzystania efektów, a także o parametrach przetwarzania charakterystycznych dla poszczególnych efektów. W rozdziale drugim opisano aplikację opracowaną w środowisku Matlab, symulującą działanie opisanych uprzednio efektów. Trzeci rozdział dotyczy systemu realizującego efekty dźwiękowe w czasie rzeczywistym. W kolejnym, czwartym rozdziale, zamieszczono wybrane wyniki testów i symulacji. W ostatnim rozdziale podsumowano wykonane prace i otrzymane wyniki.

# 1. Omówienie gitarowych efektów dźwiękowych

## 1.1. Efekty dźwiękowe

Termin „efekt dźwiękowy" należy rozumieć jako proces, którego celem i rezultatem jest modyfikacja charakterystyki dźwiękowej sygnału wejściowego [1]. W niniejszej pracy efekty dźwiękowe zostały opisane w następujących kontekstach:

- *Zjawiska fizyczne i akustyczne* - efekty dźwiękowe w wielu przypadkach odwzorowują zjawiska zachodzące w czasie propagacji fali akustycznej w przestrzeni lub wpływ analogowego układu elektronicznego na sygnał.
- *Metody cyfrowego przetwarzania sygnałów* - pozwalają one na realizację efektów dźwiękowych. W pracy opisano wykorzystane algorytmy cyfrowego przetwarzania sygnałów i sposób ich implementacji.
- *Muzyczne zastosowania* - efekty dźwiękowe są powszechnie stosowane w muzyce, zatem wskazane jest podanie możliwości ich wykorzystania oraz typowych wartości parametrów przetwarzania.

### 1.1.1. Efekty gitarowe

Wraz z upowszechnieniem się gitar elektrycznych w muzyce rozrywkowej nastała potrzeba wzbogacenia ich brzmienia poprzez zastosowanie efektów dźwiękowych. Źródeł tej potrzeby jest kilka. Gitara elektryczna nie posiada (zazwyczaj) pudła rezonansowego, którego zadaniem jest nie tylko wzmacnianie, ale również kształtowanie barwy dźwięku instrumentu. Drgania metalowych strun są konwertowane przez przetworniki elektromagnetyczne do postaci sygnału elektrycznego. Z tego powodu gitara elektryczna jest podatna na stosowanie szerokiego spektrum efektów bazujących na przekształceniu sygnału elektrycznego pochodzącego z przetworników gitary. Popularyzacja gitar elektrycznych przebiegała równolegle z intensywnym

rozwojem elektroniki, co umożliwiło tworzenie nowych efektów dźwiękowych. Jednocześnie pojawiły się potrzeby w zakresie poszukiwania nowych brzmień, często znacznie różniących się od brzmienia standardowego. Obecnie większość muzyków grających na gitarze elektrycznej stosuje jeden lub kilka efektów dźwiękowych. Najczęściej są to efekty dedykowane dla gitary elektrycznej, tzn. o parametrach przetwarzania najwłaściwszych dla tego instrumentu. W niniejszym rozdziale omówiono sześć popularnych efektów dźwiękowych charakterystycznych dla gitary elektrycznej. Przedstawione algorytmy są jednak uniwersalne i możliwe jest ich wykorzystanie również do przetwarzania dźwięku innych instrumentów. Termin „dźwiękowy efekt gitarowy" używany jest w niniejszej pracy zamiennie z krótszym, ale równie zrozumiałym określeniem „efekt gitarowy".

### 1.1.2. Wykorzystanie metod cyfrowego przetwarzania sygnałów

Rozwój multimediów spowodował rozszerzenie pola zastosowań cyfrowego przetwarzania sygnałów (CPS) dźwiękowych na sygnały inne niż sygnał mowy o ograniczonym paśmie. Obecnie głównymi obszarami, w których wykorzystuje się CPS dźwiękowych, jest kodowanie audio wysokiej jakości oraz cyfrowe wytwarzanie i modyfikowanie sygnałów muzycznych [2]. Metody cyfrowego przetwarzania sygnałów pozwalają na zastępowanie analogowych efektów dźwiękowych ich cyfrowymi odpowiednikami, ale również na tworzenie efektów bardzo trudnych lub wręcz niemożliwych do realizacji za pomocą metod analogowych. Rozwój techniki cyfrowej spowodował, że możliwe jest zarówno zastosowanie CPS do postprodukcji studyjnej dźwięku, jak i użycie efektów w czasie rzeczywistym. Jedną z ważnych zalet cyfrowego przetwarzania dźwięku jest możliwość realizacji wielu efektów przy użyciu jednego układu/urządzenia. Uproszczony zostaje również proces symulacji działania efektu przy użyciu oprogramowania komputerowego, np. Matlab. W dalszej części pracy zostało opisanych sześć efektów gitarowych:

- Delay,
- Reverb,
- Flanger,
- Tremolo,
- Overdrive,
- Pitch Shifter.

## 1.2. Delay

### 1.2.1. Wersja FIR

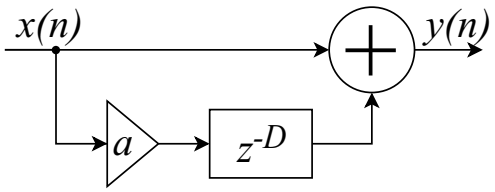
W podstawowej wersji efekt Delay symuluje echo, czyli odbicie fali dźwiękowej od odległej przeszkody o dużych rozmiarach [1]. Do sygnału wejściowego dodany zostaje sygnał opóźniony i stłumiony. W dziedzinie czasu ciągłego można to opisać za pomocą równania:

$$y(t) = x(t) + ax(t - T_{Delay}) \ , \tag{1.1}$$

gdzie $x(t)$ - sygnał wejściowy, $y(t)$ - sygnał wyjściowy, $a < 1$ - współczynnik odbicia, $T_{Delay}$ - czas opóźnienia. W dziedzinie czasu dyskretnego równanie ma postać:

$$y(n) = x(n) + ax(n - D) \tag{1.2}$$

(oznaczenia analogicznie jak w 1.1, $D$ odpowiada $T_{Delay}$). Schemat układu realizującego wersję FIR efektu Delay wygląda następująco:

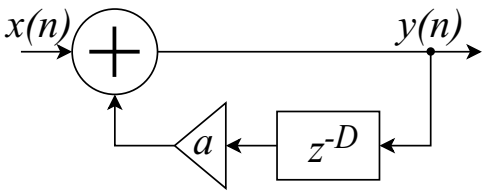



Rys. 1.1. Schemat układu realizującego efekt Delay w wersji FIR

Stosowane w praktyce wartości czasu opóźnienia zawierają się zwykle w zakresie od 50ms do pojedynczych sekund [1].

### 1.2.2. Wersja IIR

Stosuje się również rekursywną wersję efektu Delay, w której sygnał wyjściowy dołączony jest do pętli sprzężenia zwrotnego według poniższego schematu:




Rys. 1.2. Schemat układu realizującego efekt Delay w wersji IIR

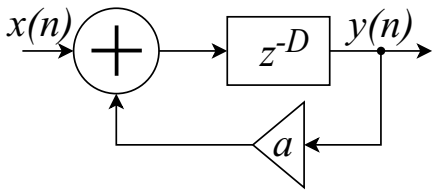
Powyższy układ można opisać równaniem:

$$y(n) = x(n) + ay(n - D) \ . \tag{1.3}$$

Tak zrealizowany efekt symuluje nieskończone odbicia od ścian o cylindrycznym kształcie [1]. Czas opóźnienia dobiera się zazwyczaj tak, żeby odpowiadał czasowi trwania np. ćwierćnuty, dzięki czemu gra na gitarze nabiera dodatkowo charakteru rytmicznego.

## 1.3. Reverb

Efekt Reverb imituje pogłos wynikający z propagacji fali akustycznej w zamkniętej przestrzeni. Zasada działania oparta jest na filtrze grzebieniowym, który zostaje zmodyfikowany tak, by miał płaską charakterystykę amplitudową [2]. Schemat filtru grzebieniowego jest następujący:

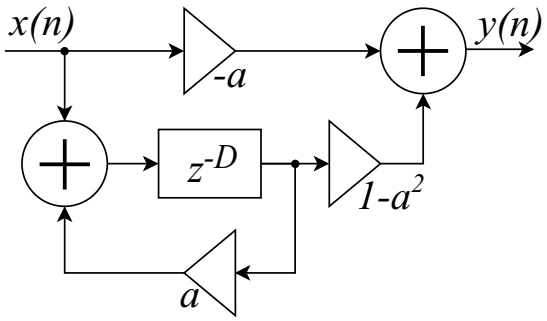



Rys. 1.3. Schemat filtru grzebieniowego

Transmitancja powyższego filtru ma postać:

$$H(z) = \frac{z^{-D}}{1 - az^{-D}} \ , \tag{1.4}$$

a jego odpowiedź impulsowa to nieskończony ciąg wykładniczo malejących impulsów oddalonych od siebie o $D$ próbek. Po modyfikacjach prowadzących do otrzymania płaskiej charakterystyki częstotliwościowej otrzymuje się następującą strukturę filtru:




Rys. 1.4. Schemat filtru wszechprzepustowego

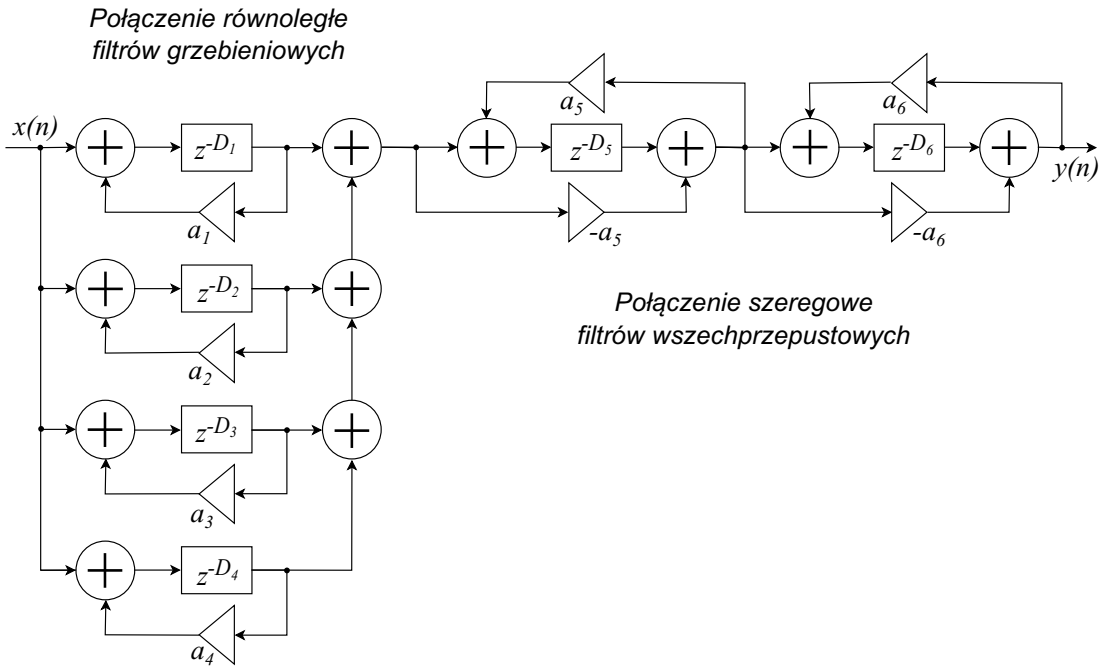
Transmitancja powyższego układu wynosi:

$$H(z) = \frac{z^{-D} - a}{1 - az^{-D}} \, . \tag{1.5}$$

Zera charakterystyki częstotliwościowej układu z rys. 1.4 są zatem takie same, jak w filtrze grzebieniowym, ale powstały filtr ma bieguny sprzężone parami. Charakterystykę częstotliwościową otrzymanego filtru opisuje równanie:

$$H(e^{j\omega}) = e^{-j\omega D} \frac{1 - ae^{+j\omega D}}{1 - ae^{-j\omega D}} \, . \tag{1.6}$$

Moduł charakterystyki częstotliwościowej filtru jest stały i równy jedności, więc układ jest filtrem wszechprzepustowym. Odpowiedź impulsowa układu składa się z nieskończonej liczby impulsów rozmieszczonych co $D$ próbek (jak w filtrze grzebieniowym). Charakterystyka fazowa filtru wszechprzepustowego jest nieliniowa, co prowadzi do dyspersji sygnału w dziedzinie czasu [2]. Mimo płaskiej charakterystyki amplitudowej filtru, jego odpowiedź brzmi bardzo podobnie, jak w przypadku filtru grzebieniowego, tzn. odczuwalna barwa dźwięku ulega zmianie. Dzieje się tak z powodu właściwości ludzkiego słuchu [2]. Aby zminimalizować wpływ efektu Reverb na barwę dźwięku, stosuje się łączenie filtrów. Filtry wszechprzepustowe łączy się szeregowo, co prowadzi do zwiększenie echa bez wpływania na charakterystykę amplitudową. Z kolei filtry grzebieniowe łączy się równolegle (z czasami opóźnień nie będącymi w prostej proporcji), dzięki czemu w otrzymanej charakterystyce amplitudowej znajdują się maksima pochodzące od każdego ze składowych filtrów grzebieniowych. Możliwa jest następująca struktura filtru realizującego efekt Reverb:




Rys. 1.5. Schemat układu realizującego efekt Reverb

Sekcja wszechprzepustowa odpowiada za wczesne echa, czasy opóźnień $D_5$, $D_6$ powinny mieć wartość pojedynczych milisekund, zaś współczynniki $a_5$, $a_6$ powinny wynosić około 0,7. W sekcji złożonej z filtrów grzebieniowych czasy opóźnień $D_1$, $D_2$, $D_3$, $D_4$ dobiera się tak, by stosunek największego do najmniejszego wyniósł około 1,5 (typowo 30-45ms) [2]. Od wyboru wartości współczynników $a_1$, $a_2$, $a_3$, $a_4$ zależy intensywność wytworzonego pogłosu [2].

## 1.4. Flanger

Efekt Flanger realizowany jest przez sumowanie sygnału oryginalnego z sygnałem opóźnionym, z czasem opóźnienia zmieniającym się okresowo. Podobnie, jak w przypadku Delay, występują dwie wersje tego efektu (o skończonej i nieskończonej odpowiedzi impulsowej). Oscylacja czasu opóźnienia powoduje powstanie charakterystycznych zniekształceń, związanych z dopplerowskim przesunięciem częstotliwości. Stosuje się niewielkie czasy opóźnienia (poniżej 15ms) i częstotliwości zmian opóźnienia rzędu pojedynczych herców [1]. Schematy układów realizujących efekt Flanger są takie same, jak w przypadku efektu Delay (rys. 1.1 i rys. 1.2), z opóźnieniem będącym funkcją czasu $D = D(n)$.

### 1.4.1. Wersja FIR

Działanie efektu Flanger w wersji ze skończoną odpowiedzią impulsową opisuje równanie:

$$y(t) = x(t) + ax(t - T_{Delay}(t)) \ , \tag{1.7}$$

gdzie $T_{Delay}(t)$ jest funkcją okresową, np.: $T_{Delay}(t) = \frac{T_{max}}{2}(1+\sin(2\pi t f_{Delay}))$, gdzie $T_{max}$ - maksymalny czas opóźnienia. W dziedzinie czasu dyskretnego powyższe równanie można przedstawić w postaci:

$$y(n) = x(n) + ax(n - D(n)) \ . \tag{1.8}$$

### 1.4.2. Wersja IIR

Wersja rekursywna efektu Flanger jest opisana równaniem:

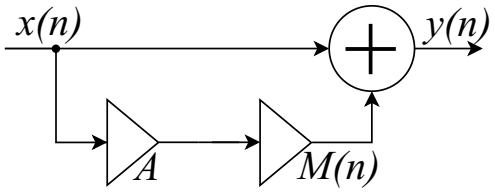
$$y(n) = x(n) + ay(n - D(n)) \ . \tag{1.9}$$

## 1.5. Tremolo

Efekt Tremolo polega na modulacji amplitudowej sygnałem o niskiej częstotliwości, jego rezultatem jest słyszalne drżenie dźwięku, czyli okresowe zmiany jego głośności [3]. Równanie opisujące efekt Tremolo ma postać:

$$y(n) = x(n)(1 + AM(n)) \ , \tag{1.10}$$

gdzie $A$ - maksymalna amplituda modulacji, $M(n)$ - funkcja modulująca [3]. Schemat układu realizującego efekt Tremolo przedstawiono na poniższym rysunku:




Rys. 1.6. Schemat układu realizującego efekt Tremolo

Typowo częstotliwość modulacji amplitudy w efekcie Tremolo wynosi od kilku do kilkunastu herców [3].

## 1.6. Overdrive

Brzmienie przesterowanej (ang. distorted) gitary elektrycznej jest jednym z najbardziej charakterystycznych elementów szeroko pojętej muzyki rockowej. Pierwot-

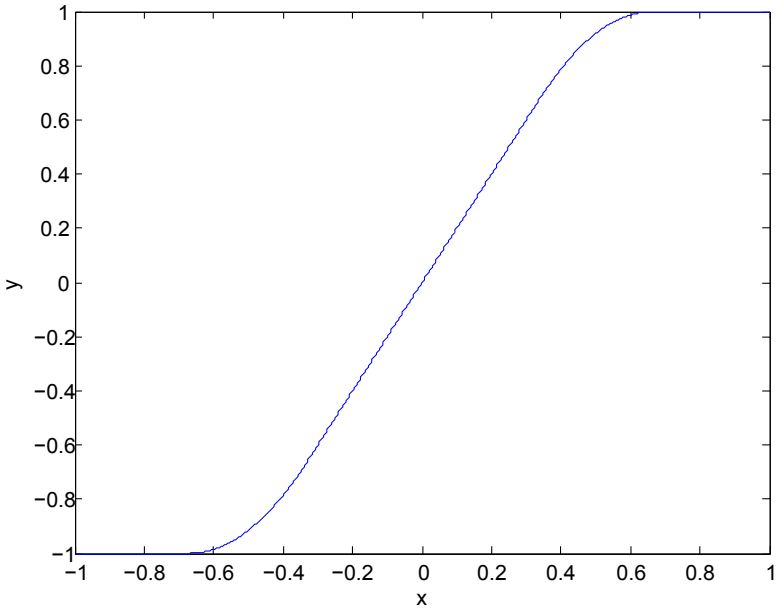
nie brzmienie to uzyskiwano przez przesterowanie elementów aktywnych wzmacniacza gitarowego, powstały również specjalne układy analogowe realizujące ten efekt. Overdrive jest efektem nieliniowym, wskutek jego działania dźwięk gitary zostaje wzbogacony przez dodanie do sygnału wyższych harmonicznych. Overdrive można zdefiniować jako stan pracy urządzenia audio (liniowego dla niskich poziomów sygnału wyjściowego), w którym wyższy poziom sygnału wejściowego powoduje przejście do nieliniowej części charakterystyki przejściowej urządzenia. Dla odpowiednio dużych amplitud sygnału wyjściowego występuje więc praca zarówno w liniowej, jak i w nieliniowej części charakterystyki urządzenia, a przejście między tymi częściami jest płynne [1].

### 1.6.1. Cyfrowy efekt Overdrive

W przypadku realizacji efektu Overdrive w dziedzinie czasu dyskretnego, nieliniowość charakterystyki przejściowej efektu jest przybliżana za pomocą odpowiedniego równania. Symulacja efektu Overdrive wymaga wykonania symetrycznego „łagodnego obcinania" (ang. soft clipping), czyli obcinania wierzchołków sygnału nie powodującego powstania ostrych krawędzi w sygnale wyjściowym [1]. Jedną z możliwości przybliżenia nieliniowości „łagodnego obcinania" jest zastosowanie równania:

$$f(x) = \begin{cases} -1 & , \ -1 \leq x \leq -\frac{2}{3} \\ \frac{-3+(2+3x)^2}{3} & , \ -\frac{2}{3} < x \leq -\frac{1}{3} \\ 2x & , \ -\frac{1}{3} < x \leq \frac{1}{3} \\ \frac{3-(2-3x)^2}{3} & , \ \frac{1}{3} < x \leq \frac{2}{3} \\ 1 & , \ \frac{2}{3} < x \leq 1 \end{cases} . \tag{1.11}$$

Dla $|x| \leq \frac{1}{3}$ sygnał przenoszony jest liniowo na wyjście. W zakresie $\frac{1}{3} < |x| \leq \frac{2}{3}$ nieliniowa charakterystyka przejściowa powoduje lekką kompresję dynamiczną sygnału [1]. Dla wyższych wartości $|x|$ sygnał na wyjściu jest stały. Charakterystykę przejściową efektu przedstawia poniższy wykres:




Rys. 1.7. Charakterystyka przejściowa efektu Overdrive

Równanie 1.11 można zmodyfikować w taki sposób, aby przy zachowaniu tego samego kształtu charakterystyki przejściowej obniżyć poziom nasycenia $y_{max}$. Zmodyfikowane, uogólnione równanie opisujące efekt Overdrive ma następującą postać:

$$f(x) = \begin{cases} -y_{max} & , \ x \leq -\frac{2}{3}y_{max} \\ y_{max}\frac{-3+(2+3x/y_{max})^2}{3} & , \ -\frac{2}{3}y_{max} < x \leq -\frac{1}{3}y_{max} \\ 2x & , \ -\frac{1}{3}y_{max} < x \leq \frac{1}{3}y_{max} \\ y_{max}\frac{3-(2-3x/y_{max})^2}{3} & , \ \frac{1}{3}y_{max} < x \leq \frac{2}{3}y_{max} \\ y_{max} & , \ \frac{2}{3}y_{max} < x \end{cases} . \tag{1.12}$$

## 1.7. Pitch Shifter

Pitch Shifter to efekt polegający na przesunięciu wysokości dźwięku bez zmiany czasu jego trwania. Sygnał akustyczny wytwarzany przez gitarę ma charakter silnie harmoniczny. Aby zachowany został tembr oryginalnego dźwięku, zarówno częstotliwość tonu podstawowego, jak i częstotliwości wyższych składowych harmonicznych powinny zostać przeskalowane przez ten sam współczynnik $\alpha$ [1]. Z tego względu mowa jest o przesunięciu wysokości dźwięku(pitch shifting), a nie częstotliwości (frequency shifting). Jako polski ekwiwalent dla słowa „pitch", używane jest określenie „wysokość", które dość dobrze oddaje akustyczny sens terminu zaczerpniętego z literatury anglojęzycznej ([2, 1]). Efekt stosowany jest w celu stworzenia harmonii z dźwiękiem oryginalnym (przez dodanie do niego odpowiedniego interwału[1]). In-

[1] w sensie muzycznym, np. kwarta, septyma

nym często spotykanym zastosowaniem jest obniżenie dźwięku o jedną oktawę, co w przypadku gitary elektrycznej pozwala na uzyskanie brzmienia gitary basowej. Mówi się wtedy o spełnianiu funkcji octavera. Jednym ze sposobów na realizację efektu Pitch Shifter jest zastosowanie algorytmu wokodera fazowego (ang. phase vocoder).

### 1.7.1. Algorytm wokodera fazowego

Algorytm wokodera fazowego jest oparty o Transformację Fouriera fragmentów sygnału o ustalonej długości (ang. Short-Time Fourier Transform - STFT). W podstawowej wersji algorytmu najpierw zmieniana jest długości sygnału bez zmiany wysokości a następnie poprzez przepróbkowanie (ang. resampling) uzyskuje się sygnał o pierwotnej długości ze zmienioną wysokością. Opis algorytmu zaczerpnięty został z [2].

#### 1.7.1.1. Quasi-stacjonarny model sinusoidalny sygnału

Algorytm wykorzystuje quasi-stacjonarny model sinusoidalny sygnału. W modelu tym sygnał jest reprezentowany jako suma sinusoid, których chwilowe amplitudy $A_i(t)$ i pulsacje $\omega_i(t)$ zmieniają się wolno w czasie.

$$x(t) = \sum_{i=1}^{I(t)} A_i(t) \exp(j\phi_i(t)) \,, \tag{1.13}$$

gdzie:

$$\phi_i(t) = \int_{-\infty}^{t} \omega_i(\tau) d\tau \,. \tag{1.14}$$

Zmienne $\phi_i(t)$ i $\omega_i(t)$ nazywane są odpowiednio chwilową fazą i chwilową pulsacją $i$-tej sinusoidy.

#### 1.7.1.2. Definicje skalowania w czasie i skalowania wysokości

W oparciu o powyższy model sygnału możliwe jest zdefiniowanie idealnego skalowania w czasie oraz skalowania wysokości.

**Skalowanie w czasie:** Celem skalowania w czasie (ang. time-scaling) jest zmiana czasu trwania sygnału bez wpływania na jego zawartość spektralną. Odwzorowanie czasu w sygnale oryginalnym na czas w sygnale zmodyfikowanym $t \rightarrow t' = T(t)$ określane jest jako funkcja wypaczenia czasu (ang. time

warping function). Funkcję $T$ można zdefiniować w postaci całkowej:

$$t \to t' = T(t) = \int_0^t \beta(\tau) d\tau \,, \tag{1.15}$$

gdzie $\beta(\tau) > 0$ jest wolnozmiennym w czasie współczynnikiem modyfikacji czasu (ang. time-modification rate). Dla stałego współczynnika $\beta(\tau) = \beta$ funkcja $T$ jest liniowa i ma postać $t' = T(t) = \beta t$. Przypadek $\beta > 1$ odpowiada zwalnianiu sygnału przez „rozciąganie" (ang. expansion) skali czasu, a przypadek $\beta < 1$ jego przyspieszaniu przez „ściśnięcie" (ang. compression) skali czasu. Dla modelu opisanego przez równanie 1.13 sygnał idealnie przeskalowany w czasie ma postać:

$$x'(t) = \sum_{i=1}^{I(T^{-1}(t'))} A_i(T^{-1}(t')) \exp(j\phi_i'(t')) \,, \tag{1.16}$$

gdzie

$$\phi_i'(t') = \int_{-\infty}^{t'} \omega_i T^{-1}(\tau) d\tau \,. \tag{1.17}$$

Z równania 1.16 wynika, że chwilowa amplituda $i$-tej sinusoidy w sygnale przeskalowanym czasowo w chwili $t'$ odpowiada jej chwilowej amplitudzie w sygnale oryginalnym w chwili $t = T^{-1}(t')$.

**Skalowanie wysokości:** Celem skalowania wysokości (ang. pitch scaling) jest zmiana zawartości spektralnej sygnału bez zmiany czasu jego trwania. Współczynnik modyfikacji wysokości (ang. pitch modification factor) $\alpha(\tau)>0$ jest w ogólności wolnozmienną funkcją czasu. W odniesieniu do modelu opisanego równaniem 1.13 sygnał o idealnie przeskalowanej wysokości ma postać:

$$x'(t') = \sum_{i=1}^{I(t')} A(t') \exp(j\phi(t')) \,, \tag{1.18}$$

gdzie:

$$\phi_i'(t') = \int_{-\infty}^{t'} \alpha(\tau)\omega_i(\tau) d\tau \,. \tag{1.19}$$

Z równania 1.18 wynika, że w zmodyfikowanym sygnale w chwili $t'$ chwilowe amplitudy składowych sinusoid są takie same, jak w sygnale oryginalnym w chwili $t = t'$, ale ich częstotliwości są przemnożone przez współczynnik $\alpha(\tau)$.

#### 1.7.1.3. Analiza

W pierwszym etapie algorytmu wokodera fazowego sygnał $x(n)$ zostaje podzielony na ramki (ang. frames) o ustalonej długości $N$. Przesunięcie między dwiema kolejnymi ramkami wynosi najczęściej $R_a = N/4$ (ramki nachodzą na siebie w 75%). Każda ramka mnożona jest przez funkcję okna $h(n)$, a następnie obliczana jest jej STFT (ang. Short-Time Fourier Transform):

$$X(t_a^u, \Omega_k) = \sum_{n=-\infty}^{\infty} h(n)x(t_a^u + n)\exp(-j\Omega_k n)\,, \tag{1.20}$$

gdzie $u$ jest indeksem ramki, $t_a^u = uR_a$ to równomiernie rozmieszczone chwile czasu odpowiadające początkom kolejnych ramek, zaś $\Omega_k = \frac{2\pi k}{N}$. Podstawiając 1.13 do 1.20 otrzymuje się:

$$X(t_a^u, \Omega_k) = \sum_{i=1}^{I(t_a^u+n)} h(n)\left(A_i(t_a^u + n)\exp(j\phi_i(t_a^u + n))\right)\exp(-j\Omega_k n)\,. \tag{1.21}$$

Przy założeniu, że okno $h(n)$ jest dostatecznie krótkie, by chwilowe pulsacje i amplitudy składowych sinusoid były stałe w czasie trwania $h$, otrzymuje się:

$$\phi_i(t_a^u + n) = \phi_i(t_a^u) + n\omega_i(t_a^u)\,. \tag{1.22}$$

Po przekształceniu równanie 1.21 można przedstawić jako:

$$X(t_a^u, \Omega_k) = \sum_{i=1}^{I(t_a^u)} A_i(t_a^u)\exp(j\phi_i(t_a^u)H(\Omega_k - \omega_i(t_a^u))\,, \tag{1.23}$$

gdzie $H(\omega)$ oznacza Dyskretną Transformatę Fouriera (DTF) okna $h(n)$. Z równania 1.23 wynika, że STFT sygnału będącego sumą sinusoid jest suma splotów $H(\omega)$ z prążkami sinusoid o pulsacjach $\omega_i(t_a^u)$, amplitudach $A_i(t_a^u)$ i fazach $\phi_i(t_a^u)$. Aby obrazy $H(\omega)$ pochodzące od poszczególnych sinusoid nie nachodziły na siebie, pulsacja odcięcia $\omega_h$ okna $h(n)$ musi być mniejsza niż odległość między dwiema sąsiednimi sinusoidami. Przy zapewnieniu tego warunku równanie 1.23 można uprościć do:

$$X(t_a^u, \Omega_k) = \begin{cases} A_i(t_a^u)\exp(j\phi(t_a^u)) & , \textit{jeśli } |\Omega_k - \omega_i(t_a^u)| \leq \omega \\ 0 & w\,p.p. \end{cases}. \tag{1.24}$$

Z równania 1.24 wynika, że STFT zwraca informację o chwilowej amplitudzie $A_i(t_a^u)$ i fazie $\phi_i(t_a^u)$ sinusoidy przypadającej na $k$-ty kanał transformaty Fouriera. Faza znana jest z dokładnością do wielokrotności $2\pi$. Aby możliwe było przeskalowanie sygnału w skali czasu (ang. time-scaling), potrzebna jest znajomość chwilowej pul-

sacji $\omega_i(t_a^u)$. Można ją estymować na podstawie STFT dwóch kolejnych ramek. Dla zadanego $k$ różnica faz początkowych pomiędzy kolejnymi STFT wynosi:

$$\Delta\phi = \phi_i(t_a^{u+1}) - \phi_i(t_a^u) = (t_a^{u+1} - t_a^u)\omega_i(t_a^u) + 2m\pi\,. \tag{1.25}$$

W powyższym równaniu przyjęto, że pulsacja $\omega_i(t_a^u)$ jest stała pomiędzy $t_a^u$ a $t_a^{u+1}$. Po podstawieniu $R(u) = t_a^{u+1} - t_a^u$ otrzymuje się ($i$-ta sinusoida przypada na $k$-ty kanał transformaty Fouriera):

$$|(\omega_i(t_a^u) - \Omega_k)R(u)| \leq \omega_h R(u)\,, \tag{1.26}$$

gdzie $\omega_h$ oznacza szerokość pasma przepustowego okna $h(n)$. Jeśli spełniony jest warunek $\omega_h R(u) < \pi$, otrzymuje się:

$$|\Delta\phi - \Omega_k R(u) - 2m\pi| < \pi \tag{1.27}$$

i tylko jedna liczba całkowita $m$ spełnia powyższą nierówność. Dla ustalonej wartości $m$, chwilową pulsację można obliczyć z równania 1.25:

$$\omega_i(t_a^u) = \Omega_k + \frac{1}{R(u)}(\Delta\phi - \Omega_k R(u) - 2m\pi)\,. \tag{1.28}$$

#### 1.7.1.4. Skalowanie w dziedzinie czasu

Podstawowa wersja algorytmu zakłada uzyskanie przesunięcia wysokości dźwięku (ang. pitch shift) poprzez rozciągnięcie sygnału w czasie (time-scaling) przy zachowaniu oryginalnej częstotliwości, a następnie przepróbkowanie (resampling). Chwile czasu odpowiadające początkom kolejnych ramek w etapie syntezy są rozmieszczone równomiernie z odstępem $t_s^{u+1} - t_s^u = R_s$. Chwile czasu z etapu analizy (punkt 1.7.1.3) oblicza się ze wzoru $t_a^u = T^{-1}(t_s^u)$. STFT sygnału przeskalowanego w czasie ma postać:

$$Y(t_s^u, \Omega_k) = \left|X(T^{-1}(t_s^u), \Omega_k)\right| \exp(j\hat{\phi}_k(t_s^u))\,, \tag{1.29}$$

gdzie:

$$\hat{\phi}_k(t_s^u) = \hat{\phi}_k(t_s^{u-1}) + (t_s^u - t_s^{u-1})\lambda_k(T^{-1}(t_s^u))\,. \tag{1.30}$$

W powyższym równaniu $\lambda_k(T^{-1}(t_s^u))$ oznacza chwilową pulsacją obliczoną w $k$-tym kanale według wzoru 1.28. Zakłada się, że $\lambda_k(t_a^u)$ jest stała w czasie pomiędzy chwilami $t_a^{u-1}$ i $t_a^u$. Można zauważyć, STFT z równania 1.29 odpowiada idealnemu

przeskalowaniu w czasie zdefiniowanemu w równaniu 1.16. Moduł zmodyfikowanego STFT w danym kanale w chwili $t_s^u$ jest taki sam, jak w oryginalnej STFT w chwili $t_a^u = T^{-1}(t_s^u)$, a faza jest zmodyfikowana tak, że w chwili $t_s^u$ chwilowa pulsacja każdej ze składowych sinusoid jest taka sama jak w oryginalnym sygnale w chwili $t_a^u = T^{-1}(t_s^u)$.

#### 1.7.1.5. Synteza

Aby uzyskać przeskalowany w czasie sygnał $y(n)$, należy poddać $Y(t_s^u, \Omega_k)$ procesowi syntezy opisanemu równaniem:

$$y(n) = \sum_{u=-\infty}^{\infty} w(n - t_s^u) \frac{1}{N} \sum_{k=0}^{N-1} Y(t_s^u, \Omega_k) \exp(j\Omega_k n - t_s^u) \,, \tag{1.31}$$

gdzie $w(n)$ nazywane jest oknem etapu syntezy. Wymnożenie fragmentu sygnału odtworzonego z $Y(t_s^u, \Omega_k)$ przez funkcję okna $w(n)$ pozwala na zmniejszenie zniekształceń związanych z dodawaniem „z zakładką" (ang. overlap add) fragmentów sygnału wg równania 1.31 [2].

#### 1.7.1.6. Podsumowanie algorytmu wokodera fazowego

Aby dokonać skalowania wysokości dźwięku, należy wykonać następujące kroki:

1. Ustawienie początkowych faz chwilowych $\hat{\phi}(0, \Omega_k) = \arg(X(0, \Omega_k))$.
2. Ustawienie początku ramki etapu syntezy $t_s^{u+1} = t_s^u + R_s$ i obliczenie początku ramki etapu analizy $t_a^u = T^{-1}(t_s^u)$.
3. Obliczenie STFT następnej ramki rozpoczynającej się w chwili $t_a^{u+1}$ i obliczenie chwilowych pulsacji w każdym kanale wg równania 1.28.
4. Obliczenie chwilowej fazy $\hat{\phi}_k(t_s^{u+1})$ z równania 1.30.
5. Odtworzenie STFT w chwili $t_s^{u+1}$ według równania 1.29.
6. Obliczenie $(u+1)$-tego fragmentu zmodyfikowanego sygnału przy użyciu równania 1.31 i powrót do punktu 2.

W ten sposób z oryginalnego sygnału $x(t)$ otrzymuje się sygnał $x'(t)$ przeskalowany w czasie ze współczynnikiem $T(t)$, takim że $\alpha(t) = \frac{dT}{dt}$. Aby powrócić do pierwotnego czasu trwania przy zmienionej wysokości dźwięku, należy przepróbkować sygnał według poniższego równania:

$$y(t) = x'(T(t)) \,. \tag{1.32}$$

Sygnał $y(t)$ ma długość oryginalnego sygnału $x(t)$, jego wysokość jest przeskalowana ze współczynnikiem $\alpha(t)$.

# 2. Symulacja gitarowych efektów dźwiękowych

## 2.1. Wprowadzenie

Pierwszym etapem realizacji pracy było wykonanie aplikacji z graficznym interfejsem użytkownika w środowisku Matlab. Pozwala ona na zastosowanie efektów gitarowych na sygnałach zapisanych w plikach WAV. W niniejszym rozdziale opisana została struktura programu, implementacja algorytmów cyfrowego przetwarzania sygnałów realizujących efekty dźwiękowe oraz sposób korzystania z aplikacji przez użytkownika.

## 2.2. Struktura programu

Opracowana aplikacja składa się z trzech grup plików. Pierwszą grupę stanowią pliki zawierające funkcje realizujące algorytmy cyfrowego przetwarzania dźwięku. Druga grupa to pliki zawierające opisy układu okien, stworzone w edytorze interfejsu graficznego GUIDE (Graphical User Interface Design Environment), będącym częścią środowiska Matlab. Trzecią grupę stanowią pliki, w których znajdują się funkcje definiujące działanie elementów składowych graficznego interfejsu użytkownika.

Funkcje realizujące algorytmy cyfrowego przetwarzania sygnałów znajdują się w następujących plikach o nazwach odpowiadających realizowanym efektom dźwiękowym:

- `delay.m`,
- `reverb.m`,
- `flanger.m`,
- `tremolo.m` ,
- `overdrive.m`,

- `pitchShifter.m`.

Przyjęto rozwiązanie, w którym uruchomienie aplikacji powoduje otwarcie okna głównego `GuitarEffectGUI`, poprzez które otwierane są okna odpowiadające realizowanym efektom dźwiękowym. Okna zostały zaprojektowane w edytorze interfejsu graficznego GUIDE, ich opisy znajdują się w plikach:

- `GuitarEffectGUI.fig`,
- `delayGUI.fig`,
- `reverbGUI.fig`,
- `flangerGUI.fig` ,
- `tremoloGUI.fig` ,
- `overdriveGUI.fig`,
- `pitchShifterGUI.fig`.

Każdemu z plików z rozszerzeniem `.fig` odpowiada plik o identycznej nazwie z rozszerzeniem `.m`, definiujący działanie elementów graficznego interfejsu użytkownika.

## 2.3. Implementacja algorytmów

W oparciu o algorytmy opisane w rozdziale 1 wykonano implementację efektów dźwiękowych. Dla zwiększenia przejrzystości kodu, każdy z plików odpowiadających za realizację efektów dźwiękowych ma podobną strukturę. Starano się również używać nazw zmiennych, które będą jak najbliższe nomenklaturze z rozdziału 1.

### Odczyt i zapis do pliku, odtwarzanie plików

Aby możliwe było przetwarzanie sygnałów zapisanych w plikach WAV, konieczne są operacje odczytu z pliku oraz zapisu do pliku. Operacje te są identyczne dla każdego z efektów. W każdej funkcji realizującej efekt dźwiękowy, odczyt jest wykonywany przed główną pętlą, w której odbywa się przetwarzanie sygnału.

Listing 2.1. Odczyt sygnału z pliku .wav

```
%odczyt z pliku
pelnaNazwa = strcat(sciezka,'\',nazwa, rozszerzenie);
[x,Fs] = wavread(pelnaNazwa);
signalLength = length(x);
```

Parametry `sciezka`, `nazwa`, `rozszerzenie` przekazywane są do funkcji realizującej efekt jako argumenty jej wywołania. W zmiennych `x`, `Fs` przechowywane są odpowiednio sygnał odczytany z oryginalnego pliku oraz częstotliwość próbkowania. Po przetworzeniu wektor zawierający sygnał wyjściowy zapisywany jest do pliku wynikowego.

Listing 2.2. Zapis sygnału do pliku .wav

```
%zapis do pliku
nazwaZapis = strcat(sciezka, '\', nazwa, '_nazwaEfektu', rozszerzenie);
wavwrite(y,Fs,nazwaZapis);
%odsluchanie rezultatu
wavplay(x,Fs);
wavplay(y,Fs);
```

Parametr `'_nazwaEfektu'` odpowiada nazwie funkcji realizującej efekt (np. `'_reverb'`). Dopisanie nazwy efektu po nazwie oryginalnego pliku pozwala na łatwą identyfikację pliku wynikowego w folderze (w tej samej lokalizacji, z której wczytano plik oryginalny). Oba pliki są automatycznie odtwarzane, aby użytkownik mógł ocenić efekt końcowy.

### 2.3.1. Delay

Funkcja `delay` realizuje efekt Delay w następujący sposób:

Listing 2.3. Plik delay.m - realizacja efektu Delay

```
function delay(sciezka, nazwa, rozszerzenie, opoznienie, sprzezenie, wersja)
    ... %wczytanie sygnalu z pliku
delayTime = floor(0.001 * opoznienie * Fs); %ms->liczba probek
a = sprzezenie;
y = zeros(signalLength,1);
for i=1:delayTime
    y(i)=x(i);
end
if (strcmp(wersja, 'FIR'))
    for i=delayTime+1:signalLength
        y(i) = (1/(1+a))*(x(i) + a*x(i-delayTime));
    end
else
    for i=delayTime+1:signalLength
        y(i) = (1/(1+a))*(x(i) + a*y(i-delayTime));
    end
end
    ... %zapis sygnalu do pliku i odsluchanie rezultatu
```

Na początku działania funkcji tworzony jest pusty wektor o długości sygnału wejściowego. Następnie w zależności od argumentu w wywołaniu funkcji realizowana jest wersja FIR lub IIR efektu. Jeśli wybrano wersję FIR, to zgodnie ze wzorem 1.2 do sygnału oryginalnego dodawana jest jego stłumiona i opóźniona wersja, w przeciwnym wypadku realizowane jest równanie 1.3 i do sygnału wejściowego dodawany jest stłumiony i opóźniony sygnał wyjściowy. Należy zauważyć, że pierwszych `delayTime` próbek sygnałów jest przepisanych wprost z wejścia na wyjście. Jest to konieczne, ponieważ w przypadku tych próbek nie można się odwołać do próbki o indeksie o `delayTime` niższym. Z kolei przemnożenie wyniku przez `(1/(1+a))` zapobiega przekroczeniu zakresu <-1, 1>, do którego muszą należeć wartości próbek sygnału zapisywanego do pliku WAV.

### 2.3.2. Reverb

Efekt Reverb jest realizowany przez funkcję `reverb`:

Listing 2.4. Plik reverb.m - realizacja efektu Reverb

```
function reverb(sciezka, nazwa, rozszerzenie, a1, a2, a3, a4, a5, a6, D1, D2,
    D3, D4, D5, D6)
    ... %wczytanie sygnalu z pliku
%ms->liczba probek
D1 = round(0.001 * D1 * Fs);
    ...
D6 = round(0.001 * D6 * Fs);
%inicjalizacja zmiennych
comb1_out = zeros(signalLength,1);
    ...
comb4_out = zeros(signalLength,1);
comb_sect_out = zeros(signalLength,1);
allpass1_out = zeros(signalLength,1);
y = zeros(signalLength,1);
%realizacja efektu Reverb
for i=max([D1,D2,D3,D4,D5,D6])+1:signalLength
    %sygnaly na wyjsciach filtrow grzebieniowych
    comb1_out(i) = x(i) + a1 * comb1_out(i-D1);
    comb2_out(i) = x(i) + a2 * comb2_out(i-D2);
    comb3_out(i) = x(i) + a3 * comb3_out(i-D3);
    comb4_out(i) = x(i) + a4 * comb4_out(i-D4);
    %suma sygnalow z wyjsc filtrow grzeb. (wejscie sekcji wszechprzepustowej)
    comb_sect_out(i) = comb1_out(i) + comb2_out(i) + comb3_out(i) + comb4_out(i
        );
    %sygnal pomiedzy pierwszym a drugim stopniem sekcji wszechprzepustowej
    allpass1_out(i) = -a5*comb_sect_out(i)
        + comb_sect_out(i-D5) + a5*allpass1_out(i-D5);
```

```
    %sygnal na wyjsciu ukladu
    y(i) = (-a6*allpass1_out(i) + allpass1_out(i-D6) + a6*y(i-D6));
end
y=y/max(abs(y));    %skalowanie sygnalu wyjsciowego do zakresu <-1,1>
    ... %zapis sygnalu do pliku i odsluchanie rezultatu
```

Zastosowano następujące oznaczenia (*vide* rys. 1.5):

- `comb1_out` ... `comb4_out` - sygnały na wyjściach filtrów grzebieniowych,
- `comb_sect_out` - sygnał na wyjściu sekcji filtrów grzebieniowych,
- `allpass1_out` - sygnał na wyjściu pierwszego z dwóch filtrów wszechprzepustowych,
- `y` - sygnał wyjściowy.

Funkcja realizuje efekt Reverb zgodnie ze schematem z rys. 1.5. Sygnały z wyjść połączonych równolegle filtrów grzebieniowych są sumowane, suma tych sygnałów trafia na kaskadę dwóch filtrów wszechprzepustowych. Wyjście drugiego z filtrów wszechprzepustowych jest jednocześnie wyjściem całego układu realizującego efekt Reverb.

### 2.3.3. Flanger

Do realizacji efektu Flanger służy funkcja `flanger`:

Listing 2.5. Plik flanger.m - realizacja efektu Flanger

```
function flanger(sciezka, nazwa, rozszerzenie,opoznienie, czestotliwosc,
    sprzezenie, wersja)
    ... %wczytanie sygnalu z pliku
timeDelayMax = floor(0.001*opoznienie*Fs); %ms->liczba probek
FreqDelayChange = czestotliwosc / Fs;
y = zeros(signalLength,1);
a = sprzezenie;
for i=1:timeDelayMax
    y(i)=x(i);
end
if (strcmp(wersja,'FIR'))
    for i=timeDelayMax+1:signalLength
        %realizacja okresowo zmiennego opoznienia
        delayTime = 1 + round(timeDelayMax/2*(1-cos(2*pi*FreqDelayChange*i)));
        y(i) =(1/(1+a))* (x(i) + a * x(i - delayTime));
    end
else
    for i=timeDelayMax+1:signalLength
```

```
        %realizacja okresowo zmiennego opoznienia
        delayTime = 1 + round(timeDelayMax/2*(1-cos(2*pi*FreqDelayChange*i)));
        y(i) =(1/(1+a))* (x(i) + a * y(i - delayTime));
    end
end
    ... %zapis sygnalu do pliku i odsluchanie rezultatu
```

W powyższej implementacji algorytmu opóźnienie zmienia się sinusoidalnie z częstotliwością `FreqDelayChange` w zakresie od 1 próbki do `timeDelayMax`+1 próbek. W zależności od wartości argumentu `wersja` realizowana jest wersja FIR (równanie 1.7) lub IIR (równanie 1.8) algorytmu.

### 2.3.4. Tremolo

Funkcją realizującą efekt Tremolo jest `tremolo`:

Listing 2.6. Plik overdrive.m - realizacja efektu Overdrive

```
function tremolo(sciezka, nazwa, rozszerzenie, czestModul, amplModul)
    ... %wczytanie sygnalu z pliku
FreqModul = czestModul / Fs;
A = amplModul;
y = zeros(signalLength, 1);

for i=1:signalLength
    y(i) = x(i)*(1 + A*(cos(2*pi*FreqModul*i)));
end
    ... %zapis sygnalu do pliku i odsluchanie rezultatu
```

Efekt Tremolo jest realizowany zgodnie z równaniem 1.10. Funkcją modulującą sygnał wejściowy jest kosinusoida o częstotliwości `czestModul` i amplitudzie `amplModul`.

### 2.3.5. Overdrive

Efekt Overdrive jest realizowany przez funkcję `overdrive`:

Listing 2.7. Plik overdrive.m - realizacja efektu Overdrive

```
function overdrive(sciezka, nazwa, rozszerzenie, poziomObcinania)
    ... %wczytanie sygnalu z pliku
y = zeros(signalLength,1);
%poziom, do ktorego bedzie ograniczony sygnal wyjsciowy
yMax = poziomObcinania;
%realizacja efektu Overdrive
for i=1:signalLength
    if ((x(i))<=-(2/3)*yMax)
```

```
        y(i)=-yMax;
    else
        if (x(i)<=-(1/3)*ySat)
            y(i)=yMax*(-3+(2+3*x(i)/yMax)^2)/3;
        else
            if (abs(x(i)<=(1/3)*yMax))
                y(i)=2*x(i);
            else
                if (x(i)<=(2/3)*yMax)
                    y(i)=yMax*(3-(2-3*x(i)/yMax)^2)/3;
                else
                    y(i)=yMax;
                end
            end
        end
    end
end
    ... %zapis sygnalu do pliku i odsluchanie rezultatu
```

Funkcja realizuje uogólnioną postać równania opisującego łagodne nasycenie (równ. 1.12). Dla sygnału wejściowego o wartościach $|x| < \frac{y_{max}}{3}$ funkcja przejściowa jest liniowa, w zakresie $\frac{y_{max}}{3} \leq |x| < \frac{2y_{max}}{3}$ nasycanie jest przybliżane przez funkcję kwadratową, dla $|x| > \frac{y_{max}}{3}$ sygnał jest obcinany do wartości $\pm y_{max}$.

### 2.3.6. Pitch Shifter

Efekt Pitch Shifter jest realizowany przez funkcję `pitchShifter`:

Listing 2.8. Plik pitchShifter.m - realizacja efektu Pitch Shifter

```
function  pitchShifter(sciezka, nazwa, rozszerzenie, liczbaPoltonow)
    ... %wczytanie sygnalu z pliku
alpha = 2.^(liczbaPoltonow/12); %poltony->wspolczynnik skalowania
N = 1024; %dlugosc ramki (okna)
Ra = N/4; %odleglosc miedzy ramkami w etapie analizy
Rs = round(alpha*Ra); %--||-- w etapie syntezy
h = hann(2*N+1); %symetryczne okno Hanninga
h = h(2:2:end);
Omega = ((0:N-1)*2*pi/N)'; %zbior pulsacji FFT N-punktowej
y = zeros(alpha*signalLength+N,1); %inicjalizacja sygnalu wyjsciowego

% REALIZACJA ALGORYTMU WOKODERA FAZOWEGO
%1. Ustawienie fazy poczatkowej
X = fft(h.*x(1:N));
phi_a = angle(X); %faza ramki (etap analizy)
phi_a_prev = phi_a; %faza poprzedniej ramki (e. analizy)
phi_s_prev = phi_a; %--||-- (e. syntezy)
```

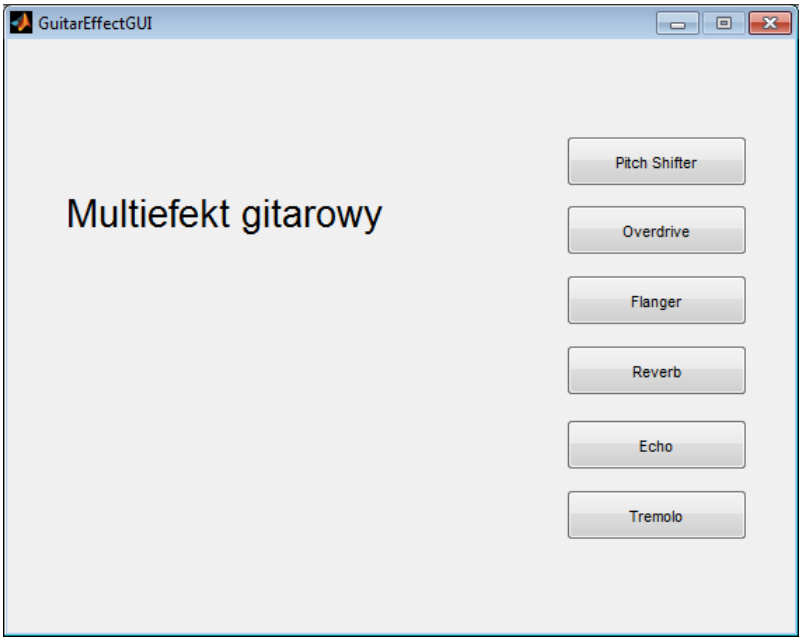
```
for u=2:signalLength/Ra-4
    %2. Obliczenie poczatkow ramek
    ts = (u-1)*Rs;
    ta = ts/alpha;
    %3. Obliczenie STFT porcji sygnalu wejsciowego
    X = fft(h.*x(ta+1:ta+N));
    phi_a = angle(X);
    A = abs(X);
    %roznica faz miedzy kolejnymi ramkami etapu analizy
    deltaPhi = phi_a - phi_a_prev;
    phi_a_prev = phi_a;
    %"odwijanie" (ang.unwrapping) fazy
    deltaPhiUnwrap = mod(deltaPhi - Omega*Ra + pi, 2*pi)- pi;
    omega_a = Omega + deltaPhiUnwrap/Ra;
    %4. Obliczenie fazy chwilowej
    phi_s = phi_s_prev + Rs*omega_a;
    phi_s_prev = phi_s;
    %5. STFT fragmentu sygnalu wyjsciowego
    Y = A.*exp(1i*phi_s);
    %6. Synteza
    y(ts+1:ts+N)=y(ts+1:ts+N)+h.*real(ifft(Y));
end
%interpolacja - przywrocenie pierwotnej dlugosci sygnalu
y=interp1((1:length(y)),y,((1:length(y)/alpha)*alpha));
```

Powyższa funkcja jest implementacją algorytmu wokodera fazowego, opisanego w punkcie 1.2.2 pracy. Kolejne operacje są wykonywane według listy kroków algorytmu wokodera fazowego zawartej w punkcie 1.7.1.6. W komentarzach w kodzie programu uwzględniono numerację z tejże listy. W czasie jednej iteracji przetwarzana jest porcja sygnału wejściowego (ramka) o długości `N`. Pierwszym etapem jest analiza sygnału, czyli obliczenie Dyskretnej Transformaty Fouriera (DTF) `X` zoknowanego sygnału. W celu wykonywania dalszych operacji używane są współrzędne biegunowe DTF zapisane w zmiennych `phi_a` (widmo fazowe) i `A` (widmo amplitudowe). Na podstawie różnicy pomiędzy widmem fazowym bieżącej ramki `phi_a`, a widmem fazowym ramki poprzedniej `phi_a_prev` estymowane są chwilowe pulsacje składowe `omega_a` sygnału wejściowego (*vide* quasi-stacjonarny model sinusoidalny opisany w punkcie 1.7.1.1). Na podstawie znajomości chwilowych pulsacji składowych `omega_a` i widma fazowego poprzedniej ramki z etapu syntezy `phi_s_prev`, a także współczynnika skalowania wysokości `alpha`, obliczane jest widmo fazowe obecnej ramki w etapie syntezy `phi_s`. Następnie zostaje odtworzone widmo częstotliwościowo-fazowe obecnej ramki w etapie syntezy `Y`. Pozwala to na syntezę fragmentu sygnału wyjściowego `y`. Należy zauważyć, że przy syntezie sygnału `y` ramki są dodawane „z zakładką” (ang. overlap/add), ze współczynnikiem

nachodzenia na siebie sąsiednich ramek (ang. overlap factor) równym `(1-alpha/4)` (ramki w etapie analizy nachodzą na siebie ze współczynnikiem 3/4). Po wykonaniu ostatniej iteracji algorytmu, w zmiennej `y` jest zapisany sygnał wyjściowy o wysokości sygnału oryginalnego i przeskalowany w czasie ze współczynnikiem `1/alpha`. By powrócić do pierwotnej długości sygnału i otrzymać sygnał o przeskalowanej wysokości, wykonywane jest przepróbkowanie sygnału za pomocą interpolacji liniowej.

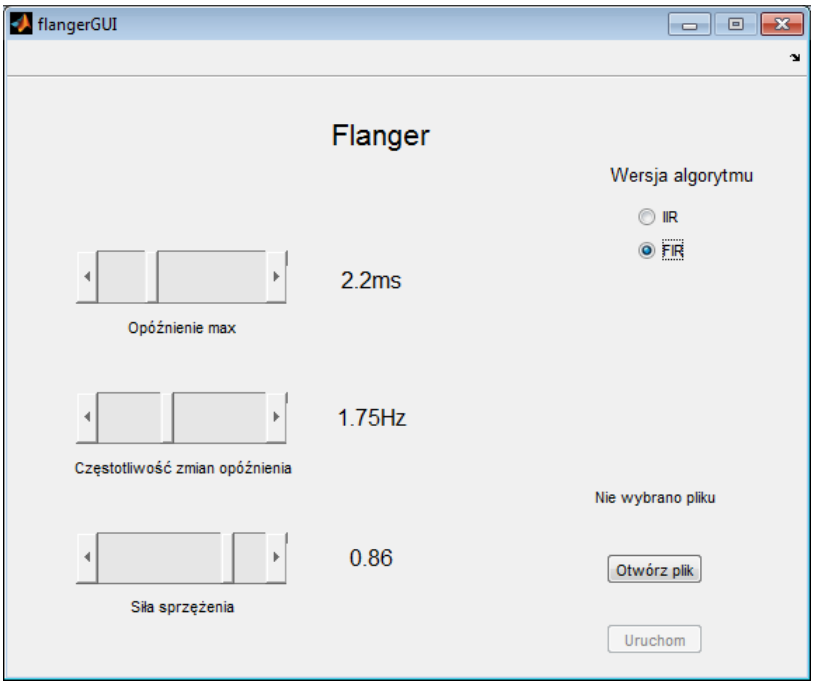
## 2.4. Obsługa aplikacji przez użytkownika

Użytkownik korzysta z aplikacji poprzez graficzny interfejs użytkownika. Uruchomienie pliku `GuitarEffectGUI.m` powoduje otwarcie głównego okna programu:



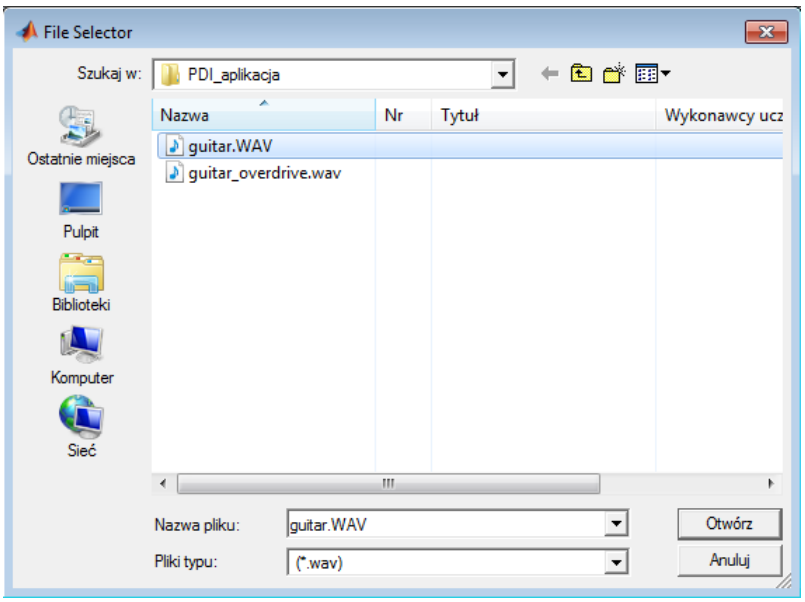

Rys. 2.1. Aplikacja GuitarEffectGUI - okno główne

Kliknięcie w przycisk z nazwą efektu powoduje otwarcie nowego okna, np. po kliknięciu w przycisk „Flanger" otwierane jest okno `FlangerGUI.fig`:




Rys. 2.2. Aplikacja GuitarEffectGUI - okno efektu Flanger

Użytkownik przy użyciu suwaków ustala parametry przetwarzania charakterystyczne dla wybranego efektu, a za pomocą przełącznika wybiera wersję algorytmu (FIR lub IIR). Przycisk „Uruchom" jest nieaktywny, dopóki nie zostanie wybrany plik WAV z sygnałem wejściowym. Po kliknięcie przycisku „Otwórz plik" otwierane jest okno wyboru pliku:




Rys. 2.3. Aplikacja GuitarEffectGUI - okno wyboru pliku

Wybranie pliku WAV i kliknięcie przycisku „Otwórz" powoduje powrót do okna efektu, w którym odblokowany zostaje przycisk „Uruchom" (*vide* rys. 2.2), kliknięcie

którego uruchamia funkcję realizującą efekt dźwiękowy. Po zakończeniu przetwarzania sygnał wyjściowy zapisywany jest do pliku. W przypadku efektu Flanger w wersji FIR i pliku wejściowego `guitar.wav`, plik wynikowy nosi nazwę `guitar_flangerFIR.wav` i jest zapisany w tym samym katalogu, z którego wczytano plik wejściowy. Użycie pozostałych efektów odbywa się w analogiczny sposób.

# 3. Realizacja efektów gitarowych w czasie rzeczywistym

## 3.1. Wprowadzenie

Głównym celem pracy było stworzenie systemu czasu rzeczywistego pozwalającego na realizację efektów gitarowych z wykorzystaniem cyfrowego przetwarzania sygnałów. System ten umożliwia podłączenie do niego sygnału z przetwornika elektromagnetycznego gitary elektrycznej, jego konwersję do postaci cyfrowej, przetwarzanie, a następnie powrót do postaci analogowej i dostarczenie sygnału elektrycznego do wzmacniacza audio lub słuchawek.

## 3.2. Elementy systemu

### 3.2.1. Zestaw ewaluacyjny STM32F4 Discovery

Przetwarzanie sygnałów o częstotliwościach z pasma akustycznego jest obecnie możliwe nie tylko przy użyciu procesorów sygnałowych, ale również mikrokontrolerów. Główną część powstałego w ramach pracy systemu czasu rzeczywistego stanowi mikrokontrolerowy zestaw ewaluacyjny STM32F4 Discovery [4], wyprodukowany przez firmę STMicroelectronics. Komponent ten odbiera cyfrowy sygnał dźwiękowy z przetwornika A/C, przetwarza go i wysyła przetworzony sygnał na przetwornik C/A. Poniżej przedstawiono najważniejsze informacje na temat zestawu STM32F4 Discovery (według [4]):

- mikrokontroler STM32F407VGT6 z 32-bitowym rdzeniem ARM Cortex M4 i jednostką zmiennoprzecinkową FPU (ang. Floating Point Unit),
- wbudowany debugger/programator ST-LINK/V2,
- zasilanie układu przez USB lub z zewnętrznego źródła 5V,
- przetwornik audio C/A CS43L22 z gniazdem mini-jack,

- przyciski: RESET i USER.

Mikrokontroler STM32F407VGT6 pracuje z maksymalną częstotliwością taktowania zegara 168MHz. Jednym z głównych powodów, dla których zdecydowano się na wybór zestawu ewaluacyjnego STM32F4 Discovery, było to, że mikrokontrolery z serii STM32F4 posiadają jednostkę zmiennoprzecinkową FPU ze zoptymalizowaną biblioteką funkcji cyfrowego przetwarzania sygnałów. Dzięki temu możliwe jest efektywne wykorzystanie mikrokontrolera do obliczeń w czasie rzeczywistym. Kolejną ważną cechą zestawu ewaluacyjnego, która zaważyła na wyborze, był wbudowany programator/debugger, pozwalający na proste i wydajne czasowo wgrywanie programu do pamięci mikrokontorlera oraz na testowanie poprawności wykonywania programu w trybie debugowania. Warto również wspomnieć niską cenę układu (około 70zł).

### 3.2.2. Moduł kodeka audio MMCodec01

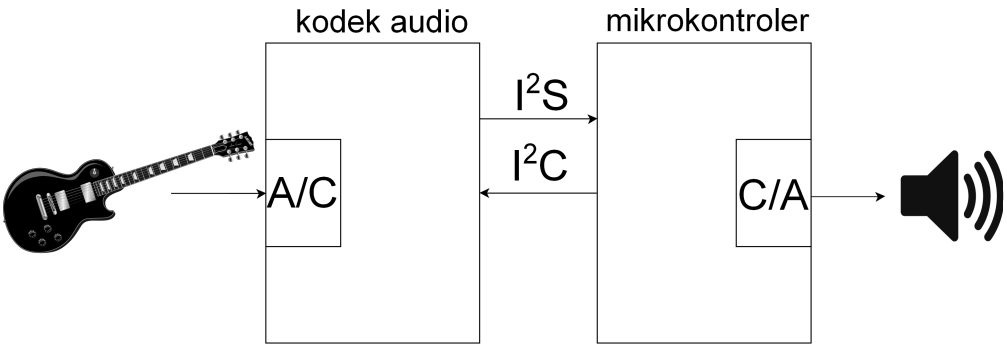
Mimo szeregu układów pomocniczych zestaw STM32F4 Discovery pozbawiony jest ważnego z punktu widzenia realizowanego systemu przetwornika audio A/C. Brak ten zdecydowano się uzupełnić przez zastosowanie modułu MMCodec01 firmy Propox [5]. Urządzenie to oparte jest o kodek audio stereo TLV320AIC23B firmy Texas Instruments. Poniżej przedstawiono podstawowe parametry kodeka TLV320AIC23B (według [6]):

- przetwornik A/C Sigma-Delta (90-dB SNR),
- przetwornik C/A Sigma-Delta (100-dB SNR),
- wejścia: liniowe i mikrofonowe,
- wyjścia: liniowe i słuchawkowe,
- obsługa interfejsów $I^2C$ (jako Slave) i $I^2S$ (jako Slave lub Master),
- częstotliwości próbkowania od 8kHz do 96kHz,
- napięcie zasilania układów cyfrowych 1,42V-3,6V,
- napięcie zasilania układów analogowych 2,7V-3,6V.

Powyższe właściwości zapewniają poprawną współpracę z układem mikroprocesora STM32F407VGT6. Moduł MMCodec01 posiada zestaw gniazd stereo typu minijack [5], co pozwala na podłączenie gitary elektrycznej za pomocą standardowych złącz. Koszt zakupu modułu MMCodec01 to około 80zł.

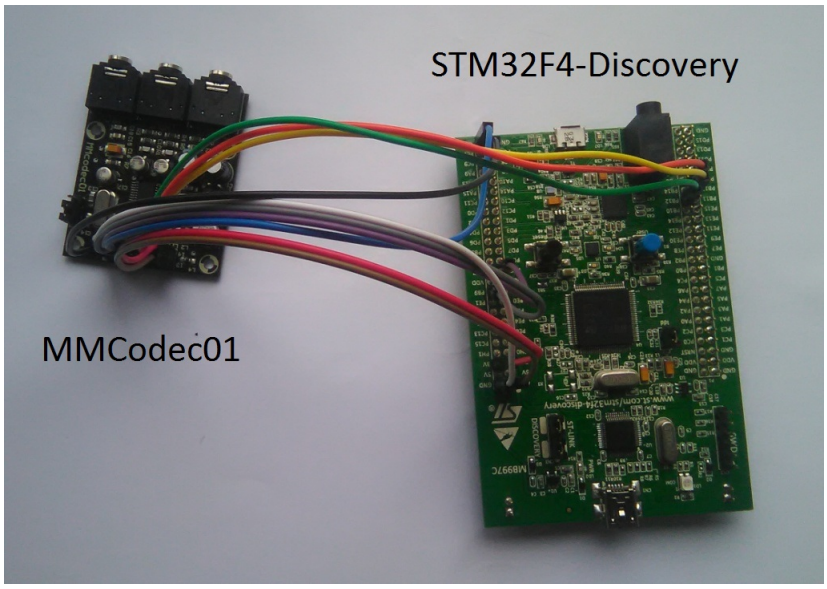
### 3.2.3. Połączenie elementów systemu

Działanie systemu wymaga wykonania odpowiednich połączeń pomiędzy jego komponentami. Schemat systemu przedstawiony jest na poniższym rysunku:




Rys. 3.1. Schemat systemu czasu rzeczywistego

Do zestawu ewaluacyjnego STM32F4-Discovery napięcie zasilania jest dostarczane przez gniazdo Mini-USB. Moduł kodeka audio MMCodec01 wymaga zasilania stabilizowanym napięciem. Jest ono dostarczane do modułu kodeka ze stabilizowanego wyjścia 3V układu STM32F4-Discovery. Połączenia pomiędzy komponentami systemu są wykonane za pomocą przewodów typu gold-pin. Poniżej przedstawiono fotografię systemu:




Rys. 3.2. Połączenie elementów systemu

## 3.3. Oprogramowanie

Oprogramowanie mikrokontrolera zostało stworzone w środowisku IAR Embedded Workbench.

### 3.3.1. Struktura projektu

Projekt `GuitarEffect.ewp` składa się z pięciu grup plików źródłowych:

**CMSIS-DSP** (Cortex Microcontroller Software Interface Standard-Digital Signal Processing) - zoptymalizowane biblioteki operacji cyfrowego przetwarzania sygnałów.

**EWARM** (IAR Embedded Workbench for ARM) - zawiera plik uruchomieniowy (ang. startup file) z tablicą wektorów dla środowiska IAR Embedded Workbench.

**STM32F4-Discovery** - biblioteka funkcji służących do obsługi układów pomocniczych wchodzących w skład zestawu ewaluacyjnego STM32F4 Discovery.

**STM32F4xxStdPeriph_Driver** - sterowniki obsługi układów peryferyjnych mikrokontrolera STM32F407VGT6.

**User** - pliki, w które ingeruje programista.

### 3.3.2. Konfiguracja kodeka audio

Aby realizować próbkowanie dźwięku, konieczne jest skonfigurowanie kodeka audio TLV320AIC23B. W tym celu użyty został interfejs $I^2C$ (Inter-Integrated Circuit) [7]. Jest to dwukierunkowa szeregowa magistrala, którą obsługuje zarówno mikrokontroler STM32F407VGT6 [8], jak i kodek audio TLV320AIC23B [6]. Magistrala I2C ma dwie linie: linię danych SDA i linię zegarową SCL [7]. Mikrokontroler musi pracować w trybie Master, ponieważ to on ma przesłać do kodeka polecenia konfigurujące. Konieczne jest zatem uruchomienie układu $I^2C$ w mikrokontrolerze. Do tego celu wykorzystano funkcje z biblioteki sterowników układów peryferyjnych `STM32F4xxStdPeriph_Driver`. Konfiguracja i uruchomienie układu $I^2C$ jest realizowane przez funkcję `I2C_Config()` zapisaną w pliku `aic23.c`:

Listing 3.1. Konfiguracja ukladu I2C mikrokontrolera

```
void I2C_Config(void)
{
    GPIO_InitTypeDef      GPIO_InitStructure;
    GPIO_StructInit(&GPIO_InitStructure);
    I2C_InitTypeDef       I2C_InitStructure;
    //wlaczenie zegara dla I2Cx
    I2Cx_CLK_INIT(I2Cx_CLK, ENABLE);
    //uruchomienie zagera dla portow GPIO
    RCC_AHB1PeriphClockCmd(I2Cx_SCL_GPIO_CLK | I2Cx_SDA_GPIO_CLK, ENABLE);
```

```
    //konfiguracja funkcji alternatywnych wyprowadzen
    GPIO_PinAFConfig(I2Cx_SCL_GPIO_PORT, I2Cx_SCL_SOURCE, I2Cx_SCL_AF);
    GPIO_PinAFConfig(I2Cx_SDA_GPIO_PORT, I2Cx_SDA_SOURCE, I2Cx_SDA_AF);
    //ustawienie parametrow wyprowadzen
    GPIO_InitStructure.GPIO_Mode          = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed         = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType         = GPIO_OType_OD;
    GPIO_InitStructure.GPIO_PuPd          = GPIO_PuPd_NOPULL;
    //inicjalizacja wyprowadzenia dla linii zegarowej
    GPIO_InitStructure.GPIO_Pin           = I2Cx_SCL_PIN;
    GPIO_Init(I2Cx_SCL_GPIO_PORT, &GPIO_InitStructure);
    //inicjalizacja wyprowadzenia dla linii danych
    GPIO_InitStructure.GPIO_Pin           = I2Cx_SDA_PIN;
    GPIO_Init(I2Cx_SDA_GPIO_PORT, &GPIO_InitStructure);
    // Deinicjalizacja i ustawienie parametrow ukladu I2C
    I2C_DeInit(I2Cx);
    I2C_InitStructure.I2C_ClockSpeed          = 100000;
    I2C_InitStructure.I2C_Mode                = I2C_Mode_I2C;
    I2C_InitStructure.I2C_DutyCycle           = I2C_DutyCycle_2;
    I2C_InitStructure.I2C_OwnAddress1         = MCU_I2C_ADDRESS;
    I2C_InitStructure.I2C_Ack                 = I2C_Ack_Enable;
    I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    //Inicjalizacja ukladu I2C
    I2C_Init(I2Cx, &I2C_InitStructure);
}
```

W mikrokontrolerze STM32F407VGT6 wyprowadzenia mogą pracować jako wejścia, wyjścia, w trybie analogowym lub być podłączone do jednej z szesnastu funkcji alternatywnych [9]. Dzięki ostatniej możliwości wyprowadzenia można skonfigurować jako wejścia/wyjścia układu $I^2C$ [8]. Aby funkcja `I2C_Config()` była uniwersalna, korzysta ona z nazw zdefiniowanych w pliku nagłówkowym `aic23.h`. Przykładowo definicje parametrów dla linii zegarowej SCL są następujące:

Listing 3.2. Definicje parametrow wyprowadzenia dla linii SCL

```
#define I2Cx_SCL_PIN           GPIO_Pin_8 //Numer wyprowadzenia
#define I2Cx_SCL_GPIO_PORT     GPIOB //Port
#define I2Cx_SCL_GPIO_CLK      RCC_AHB1Periph_GPIOB
#define I2Cx_SCL_SOURCE        GPIO_PinSource8
#define I2Cx_SCL_AF            GPIO_AF_I2C1 //Funkcja alternatywna
```

Dzięki korzystaniu z definicji zmiana układu $I^2C$ lub wyprowadzeń nie wymaga ingerowania w funkcję `void I2C_Config()`, tylko zmodyfikowania definicji w pliku nagłówkowym `aic23.h`. Po wywołaniu funkcji `void I2C_Config()` układ $I^2C$ w mikrokontrolerze jest skonfigurowany i można za jego pomocą przesłać do kodeka audio TLV320AIC23B instrukcje konfigurujące jego działanie. Procedura jest następująca: urządzenie nadrzędne $I^2C$ (mikrokontroler) generuje warunek rozpoczęcia transmisji

danych i wystawia na linię danych 7-bitowy adres urządzenia podrzędnego (kodek) i bit R/W, a następnie dane w ośmiobitowych blokach. Po odebraniu każdego bajtu urządzenie podrzędne wystawia sygnał ACK na linię danych Po zakończeniu transferu urządzenie nadrzędne generuje warunek zakończenia transmisji. Kodek audio TLV320AIC23B ma 11 rejestrów 9-bitowych o 7-bitowych adresach [6]. Skonfigurowanie jednego rejestru wymaga wystawienia przez urządzenie nadrzędne adresu $I^2C$ kodeka i przesłanie dwóch bajtów danych (adres rejestru + dane do zapisania w rejestrze). Konfiguracja kodeka audio jest realizowana przez funkcję `I2C_Ctrl_Init()`. Poniżej przedstawiono funkcję służącą do przesłania danych do rejestrów kodeka TLV320AIC23B:

Listing 3.3. Konfiguracja kodeka TLV320AIC23B za pomocą interfejsu I2C

```
void I2C_Ctrl_Init(void) {
    uint8_t CodecCommandBuffer[2];
    // reset kodeka
    CodecCommandBuffer[0] = REG_RESET_RGSTR;
    CodecCommandBuffer[1] = 0x00;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    // wylacz ADC, DAC, IN, OUT, MIC, wlacz zasilanie ukladu i CLK
    CodecCommandBuffer[0] = REG_POWER_DOWN_CTR;
    CodecCommandBuffer[1] = 0x3F;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //glosnosc lewego wejscia liniowego: +12dB
    CodecCommandBuffer[0] = REG_LEFT_LINE_IN_VOL;
    CodecCommandBuffer[1] = 0x1F;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //glosnosc prawego wejscia liniowego: +12dB
    CodecCommandBuffer[0] = REG_RIGHT_LINE_IN_VOL;
    CodecCommandBuffer[1] = 0x1F;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //lewe wyjscie sluchawkowe: 0dB
    CodecCommandBuffer[0] = REG_LEFT_HEAD_VOL;
    CodecCommandBuffer[1] = 0xF9;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //prawe wyjscie sluchawkowe: 0dB
    CodecCommandBuffer[0] = REG_RIGHT_HEAD_VOL;
    CodecCommandBuffer[1] = 0xF9;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //DAC: Off, Bypass: Disabled, ADC Input: Line, MIC: Muted
    CodecCommandBuffer[0] = REG_ANALOG_AUDIO_PATH_CTR;
    CodecCommandBuffer[1] = 0x02;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //DAC soft mute: Enabled, ADC HP filter: Enabled
    CodecCommandBuffer[0] = REG_DIGITAL_AUDIO_PATH_CTR;
    CodecCommandBuffer[1] = 0x08;
```

```
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //Interfejs: I2S, 16bit/probke
    CodecCommandBuffer[0] = REG_DIGITAL_AUDIO_INTERFACE;
    CodecCommandBuffer[1] = 0x02;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //fs=48kHz, nadprobkowanie 256x
    CodecCommandBuffer[0] = REG_SAMPLE_RATE_CTR;
    CodecCommandBuffer[1] = 0x00;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //aktywowanie interfejsu cyfrowego
    CodecCommandBuffer[0] = REG_DIGITAL_INTERFACE_ACTIVATION;
    CodecCommandBuffer[1] = 0x01;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
    //Device power: On, ADC: On, CLK: On, Line Input: On
    CodecCommandBuffer[0] = REG_POWER_DOWN_CTR;
    CodecCommandBuffer[1] = 0x2A;
    I2C_Ctrl_Send(CodecCommandBuffer, 2);
}
```

Do przesłania danych za pomocą interfejsu $I^2C$ służy funkcja `I2C_Ctrl_Send()`, korzystająca z funkcji biblioteki Standard Peripheral Library:

Listing 3.4. Transmisja danych przez interfejs I2C

```
void I2C_Ctrl_Send(uint8_t controlBytes[], uint8_t numBytes)
{
    uint8_t bytesSent=0;
    while (I2C_GetFlagStatus(I2Cx, I2C_FLAG_BUSY)) { } //oczekiwanie na
        zwolnienie ukladu I2C
    //wygenerowanie warunku rozpoczecia transmisji
    I2C_GenerateSTART(I2Cx, ENABLE);
    while (!I2C_GetFlagStatus(I2Cx, I2C_FLAG_SB)) { }
    //wystawienie adresu ukladu podrzednego
    I2C_Send7bitAddress(I2Cx, KODEK_I2C_ADDRESS, I2C_Direction_Transmitter);
    while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED)) {
         }
    //przeslanie numBytes bajtow danych
    while (bytesSent < numBytes)
    {
        I2C_SendData(I2Cx, controlBytes[bytesSent]);
        bytesSent++;
        while (!I2C_CheckEvent(I2Cx, I2C_EVENT_MASTER_BYTE_TRANSMITTING)) { }
    }
    while(!I2C_GetFlagStatus(I2Cx, I2C_FLAG_BTF)) { }
    //wygenerowanie warunku zakonczenia transmisji
    I2C_GenerateSTOP(I2Cx, ENABLE);
}
```

W analogiczny sposób jest konfigurowany przetwornik audio C/A CS43L22, będący częścią zestawu ewaluacyjnego STM32F4-Discovery. Funkcje konfigurujące przetwornik CS43L22 są zapisane w pliku `cs43.c`, noszą nazwy: `cs43_I2C_Config(void)`, `cs43_I2C_Ctrl_Init()`, `cs43_I2C_Ctrl_Send()` i mają konstrukcję zbliżoną do funkcji konfigurujących kodek TLV320AIC23B.

### 3.3.3. Konfiguracja interfejsu I²S

Zarówno mikrokontroler STM32F407VGT6, jak i kodek audio TLV320AIC23B oraz przetwornik audio C/A CS43L22, obsługują interfejs I²S [8, 6, 10]. I²S (Inter-IC Sound) [11] jest interfejsem szeregowym, służącym do przesyłania danych audio pomiędzy układami scalonymi. Wykorzystuje trzy linie: linię zegarową SCK, linię wyboru kanału WS oraz linię danych SD [11]. W mikrokontrolerze STM32F407VGT6 interfejs I²S jest obsługiwany przez układy peryferyjne SPI/I2S [8]. Zdecydowano się na konfigurację, w której układy peryferyjne SPI/I2S mikroprocesora pełnią funkcję urządzeń nadrzędnych (Master) przy komunikacji z kodekiem audio TLV320AIC23B i przetwornikiem C/A CS43L22. Konfiguracja układu SPI/I2S do komunikacji z kodekiem TLV320AIC23B jest wykonywana przez funkcję `I2S_Config()`:

Listing 3.5. Konfiguracja interfejsu I2S

```
void I2S_Config(void)
{
    //konfiguracja przerwan od I2Sx
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    NVIC_InitStructure.NVIC_IRQChannel = I2Sx_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //struktury sluzace do inicjalizacji peryferiow
    GPIO_InitTypeDef      GPIO_InitStructure;
    GPIO_StructInit(&GPIO_InitStructure);
    I2S_InitTypeDef       I2S_InitStructure;
    //wlaczenie zegara dla I2Sx
    I2Sx_CLK_INIT(I2Sx_CLK, ENABLE);
    RCC_I2SCLKConfig(RCC_I2S2CLKSource_PLLI2S);
    RCC_PLLI2SCmd(ENABLE);
    while (!RCC_GetFlagStatus(RCC_FLAG_PLLI2SRDY)) { }
    //wlaczenie zegarow dla portow we/wy
    RCC_AHB1PeriphClockCmd(I2Sx_WS_GPIO_CLK | I2Sx_CK_GPIO_CLK |
        I2Sx_SD_GPIO_CLK | I2Sxext_SD_GPIO_CLK | I2Sx_MCK_GPIO_CLK, ENABLE);
```

```
    //Konfiguracja AF (Alternate Function) wyprowadzen
    GPIO_PinAFConfig(I2Sx_WS_GPIO_PORT, I2Sx_WS_SOURCE, I2Sx_WS_AF);
    GPIO_PinAFConfig(I2Sx_CK_GPIO_PORT, I2Sx_CK_SOURCE, I2Sx_CK_AF);
    GPIO_PinAFConfig(I2Sx_MCK_GPIO_PORT, I2Sx_MCK_SOURCE, I2Sx_MCK_AF);
    GPIO_PinAFConfig(I2Sx_SD_GPIO_PORT, I2Sx_SD_SOURCE, I2Sx_SD_AF);
    //Konfiguracja parametrow wyprowadzen
    GPIO_InitStructure.GPIO_Mode  = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_DOWN;
    // inicjalizacja wyprowadzenia I2S WS
    GPIO_InitStructure.GPIO_Pin = I2Sx_WS_PIN;
    GPIO_Init(I2Sx_WS_GPIO_PORT, &GPIO_InitStructure);
    // inicjalizacja wyprowadzenia I2S CK
    GPIO_InitStructure.GPIO_Pin = I2Sx_CK_PIN;
    GPIO_Init(I2Sx_CK_GPIO_PORT, &GPIO_InitStructure);
    // inicjalizacja wyprowadzenia I2S MCK
    GPIO_InitStructure.GPIO_Pin = I2Sx_MCK_PIN;
    GPIO_Init(I2Sx_MCK_GPIO_PORT, &GPIO_InitStructure);
    // inicjalizacja wyprowadzenia I2S SD
    GPIO_InitStructure.GPIO_Pin = I2Sx_SD_PIN;
    GPIO_Init(I2Sx_SD_GPIO_PORT, &GPIO_InitStructure);
    // Konfiguracja parametrow ukladu I2Sx
    SPI_I2S_DeInit(I2Sx);
    I2S_InitStructure.I2S_AudioFreq  = I2S_AudioFreq_48k;
    I2S_InitStructure.I2S_Standard   = I2S_Standard_Phillips;
    I2S_InitStructure.I2S_MCLKOutput = I2S_MCLKOutput_Enable;
    I2S_InitStructure.I2S_CPOL       = I2S_CPOL_High;
    I2S_InitStructure.I2S_DataFormat = I2S_DataFormat_16b;
    I2S_InitStructure.I2S_Mode       = I2S_Mode_MasterRx;
    //inicjalizacja I2Sx
    I2S_Init(I2Sx, &I2S_InitStructure);
    //uruchomienie ukladu I2Sx
    I2S_Cmd(I2Sx, ENABLE);
}
```

Podobnie jak opisane wcześniej funkcje `I2C_Config()`, `I2C_Ctrl_Init()` oraz `I2C_Ctrl_Send()`, funkcja `I2S_Config()` korzysta z nazw zdefiniowanych w pliku nagłówkowym `aic23.h`. Dzięki temu możliwa jest zmiana układu SPI/I2S, z którego korzysta funkcja `I2S_Config()` lub użytych wyprowadzeń bez ingerowania w kod samej funkcji. W połączeniu pomiędzy układami $I^2S$ mikrokontrolera i kodeka audio wykorzystana została dodatkowa linia MCK (Master Clock) wymagana przez kodek TLV320AIC23B [6] i dostarczana przez układ SPI/I2S mikrokontrolera STM32F407VGT6 [8].

### 3.3.4. Konfiguracja przerwań

Dane z układu SPI/I2S mogą być odczytywane w trzech trybach [8]:

- Sprawdzanie znacznika rejestru odbiorczego w nieskończonej pętli, odczyt, gdy znacznik informuje o pojawieniu się danych w buforze.
- Generowanie przerwania przez układ SPI/I2S, gdy w buforze odbiorczym pojawiają się dane i odczyt danych w funkcji obsługi przerwania.
- Wykorzystanie mechanizmu bezpośredniego dostępu do pamięci (ang. Direct Memory Access - DMA).

Pierwszy tryb jest najprostszy, jednak bardzo nieefektywny, ponieważ absorbuje procesor przez konieczność sprawdzania, czy w buforze odbiorczym układu SPI/I2S pojawiły się dane. Wykorzystanie mechanizmu przerwań zwalnia procesor z konieczności odpytywania (ang. polling) układu SPI/I2S. Po pojawieniu się danych w buforze odbiorczym układ SPI/I2S generuje żądanie obsługi przerwania, które zostaje obsłużone przez procesor. W przypadku strumienia danych o dużym natężeniu możliwe jest wykorzystanie mechanizmu DMA, który pozwala na przesyłanie danych pomiędzy układem peryferyjnym a pamięcią bez udziału procesora. Wadą takiego rozwiązania jest generowanie opóźnienia wynikającego z rozmiaru bufora odbiorczego. W stworzonym systemie zdecydowano się skorzystać z mechanizmu przerwań. Konfiguracja przerwań jest wykonywana przez funkcję `NVIC_Config()`:

Listing 3.6. Konfiguracja przerwania od układu SPI/I2S

```
void NVIC_Config(void)
{
    //konfiguracja priorytetu przerwania od ukladu I2Sx
    NVIC_InitTypeDef NVIC_InitStructure;
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);
    NVIC_InitStructure.NVIC_IRQChannel = I2Sx_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //wlaczenie przerwania przy pojawieniu sie danych w buforze wejsciowym
    SPI_I2S_ITConfig(I2Sx, SPI_I2S_IT_RXNE, ENABLE);
}
```

### 3.3.5. Realizacja efektów dźwiękowych

Realizacja efektów dźwiękowych wymaga odebrania danych przesłanych przez przetwornik A/C kodeka audio TLV320AIC23B, przetworzenie ich i wysłanie danych

przetworzonych do przetwornika C/A CS43L22. Operacje te są wykonywane w funkcji obsługi przerwania generowanego przez układ SPI/I2S mikrokontrolera. Aby możliwa była realizacja więcej niż jednego efektu dźwiękowego, funkcja obsługi przerwania zawiera warunek wielokrotnego wyboru `switch`:

Listing 3.7. Konfiguracja przerwania od układu SPI/I2S

```
void SPI2_IRQHandler(void)
{
    switch (option)
    {
        //tryb talk-through (brak modyfikacji sygnalu)
        case 0:
        {
            //lewy kanal
            if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
            {
                //odczytanie danych z bufora odbiorczego
                x = SPI_I2S_ReceiveData(I2Sx);
                //oczekiwanie na zwolnienie bufora nadawczego
                while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET)
                    {}
                //wyslanie danych wyjsciowych do przetwornika C/A
                SPI_I2S_SendData(I2Sout, x);
            }
            //prawy kanal
            else
            {
                //oproznienie bufora, probka nie jest wykorzystana (sygnal mono
                    )
                right_in =  SPI_I2S_ReceiveData(I2Sx);
                while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET)
                    {}
                SPI_I2S_SendData(I2Sout, x);
            }
        }
        break;
        case 1:
        {
            ... //Efekt 1.
        }
        break;
        case 2:
        {
            ... //Efekt 2.
        }
        break;
```

```
            ... //dalsze efekty
    }
}
```

Powyższy fragment programu przedstawia sposób realizacji wielu efektów dźwiękowych w jednej funkcji obsługi przerwania. Zawiera również implementację trybu *talk through*, czyli przekazywania danych z przetwornika A/C na przetwornik C/A bez modyfikacji. Pozostałe efekty zostały zaimplementowane w oparciu o szkielet, który stanowi tryb *talk through*.

#### 3.3.5.1. Delay FIR

Poniższy listing przedstawia realizację nierekursywnej wersji efektu Delay:

Listing 3.8. Realizacja efektu Delay FIR

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    left_in[i] = SPI_I2S_ReceiveData(I2Sx);
    //sumowanie sygnalu oryginalnego i sygnalu opoznionego
    left_out[i] = left_in[i] + DELAY_FIR_FBCK*left_in[(i + BUFFER_SIZE -
        delayTimeFIR) % BUFFER_SIZE];
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
}
else{
    right_in = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
    i = (i + BUFFER_SIZE + 1) % (BUFFER_SIZE);
}
```

We fragmencie programu przedstawionym w powyższym listingu wykorzystane są zmienne `left_in[]`, `left_out[]`, przechowujące `BUFFER_SIZE` ostatnich próbek odpowiednio sygnału wejściowego i wyjściowego. Oba sygnały zapisywane są w tablicach w sposób cykliczny, tzn. po zapełnieniu tablicy kolejna próbka jest zapisywana jako pierwszy element tablicy itd. Indeks próbki opóźnionej (według wzoru 1.2) jest w związku z tym obliczany z wykorzystaniem operacji modulo. Ponieważ dostarczany z gitary sygnał jest monofoniczny, wejścia kanału lewego i prawego przetwornika A/C kodeka audio są fizycznie zwarte. W związku z tym nie jest konieczne przetwarzanie próbek z obu kanałów. Zmienna `right_in` służy wyłącznie do opróżniania bufora odbiorczego układu SPI/I2S w mikrokontrolerze. Do przetwornika C/A w prawym kanale wysyłany jest sygnał zapisany w zmiennej `left_out[]`. Licznik próbki `i` jest inkrementowany po wysłaniu próbek do obu kanałów przetwornika C/A.

#### 3.3.5.2. Delay IIR

Rekursywna wersja efektu Delay jest realizowany w następujący sposób:

Listing 3.9. Realizacja efektu Delay IIR

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    left_in[i] = SPI_I2S_ReceiveData(I2Sx);
    //sumowanie sygnalu oryginalnego i opoznionego sygnalu wyjsciowego
    left_out[i] = left_in[i] + DELAY_IIR_FBCK*left_out[(i + BUFFER_SIZE -
        delayTimeIIR) % BUFFER_SIZE];
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
}
else{
    right_in = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
    i = (i + BUFFER_SIZE + 1) % (BUFFER_SIZE);
}
```

Powyższy listing przedstawia realizację równania opisującego efekt Delay IIR (równanie 1.3). Zarówno wykorzystane zmienne, jak i sposób korzystania z nich są takie same, jak w przypadku efektu Delay FIR, jednak do sygnału wejściowego dodany jest opóźniony i stłumiony sygnał z wyjścia (*vide* rys. 1.2).

#### 3.3.5.3. Flanger FIR

Efekt Flanger realizowany jest przez fragment programu przedstawiony poniżej:

Listing 3.10. Realizacja efektu Flanger FIR

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    left_in[i] = SPI_I2S_ReceiveData(I2Sx);
    //realizacja okresowo zmiennego opoznienia
    delayFlangerFIR = 1 + (maxDelayFlangerFIR/2 * (1 + arm_cos_f32(2*PI *
        freqFlangerFIR * countFlangerFIR)));
    //sumowanie sygnalu oryginalnego z sygnalem opoznionym
    left_out[i] = left_in[i] + FLANGER_FIR_FBCK * left_in[(i + BUFFER_SIZE -
        delayFlangerFIR) % BUFFER_SIZE];
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
}
else
{
    right_in = SPI_I2S_ReceiveData(I2Sx);
```

```
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
    //inkrementacja licznika probki
    i = (i + BUFFER_SIZE + 1) % (BUFFER_SIZE);
    //inkrementacja licznika okresowego opoznienia
    countFlangerFIR = (countFlangerFIR + periodFlangerFIR + 1) % (
        periodFlangerFIR);
}
```

Powyższy fragment programu realizuje równanie 1.7 opisujące nierekursywną wersję efektu Flanger. Zmienne opóźnienie jest obliczane za pomocą funkcji `arm_cos_f32()` z biblioteki CMSIS DSP i zaokrąglane do całkowitej liczby próbek. Należy zauważyć, że przy obliczaniu opóźnienia jest używany licznik `countFlangerFIR`, który zmienia się w zakresie od 0 do `periodFlangerFIR`-1, podczas gdy licznik pozycji próbki w tablicy zmienia się od 0 do `BUFFER_SIZE`. Zabieg taki jest konieczny, ponieważ rozmiar tablicy nie musi odpowiadać okresowi zmian opóźnienia, ani być jego całkowitą wielokrotnością i niezastosowanie oddzielnego licznika mogłoby prowadzić do skokowych zmian opóźnienia.

#### 3.3.5.4. Flanger IIR

Efekt Flanger w wersji rekursywnej jest realizowany przez przedstawiony poniżej fragment programu:

Listing 3.11. Realizacja efektu Flanger IIR

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    left_in[i] = SPI_I2S_ReceiveData(I2Sx);
    //realizacja okresowo zmiennego opoznienia
    delayFlangerIIR = 1 + (maxDelayFlangerIIR/2 * (1 + arm_cos_f32(2*PI *
        freqFlangerIIR * countFlangerIIR)));
    //sumowanie sygnalu oryginalnego z sygnalem opoznionym
    left_out[i] = left_in[i] + FLANGER_IIR_FBCK * left_in[(i + BUFFER_SIZE -
        delayFlangerIIR) % BUFFER_SIZE];
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
}
else
{
    right_in = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, left_out[i]);
    //inkrementacja licznika probki
    i = (i + BUFFER_SIZE + 1) % (BUFFER_SIZE);
    //inkrementacja licznika okresowego opoznienia
```

```
    countFlangerIIR = (countFlangerIIR + periodFlangerIIR + 1) % (
        periodFlangerIIR);
}
```

Powyższy fragment programu realizuje wersję IIR efektu Flanger opisaną równaniem 1.8. Podobnie, jak w przypadku nierekursywnej wersji efektu, zmienne opóźnienie jest obliczane przy użyciu funkcji `arm_cos_f32()`.

#### 3.3.5.5. Overdrive

Za realizację efektu Overdrive odpowiada fragment programu przedstawiony poniżej:

Listing 3.12. Realizacja efektu Overdrive

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    x = SPI_I2S_ReceiveData(I2Sx);
    if (x < -2*yMax/3)
        y = -yMax;          //obcinanie do -yMax
    else if (x < -yMax/3)
        y = yMax/3 + 4*x + 3*x*x/yMax; //nasycanie - funkcja kwadratowa
    else if (x < yMax/3)
        y = 2*x;          // odcinek liniowy (male amplitudy)
    else if (x < 2*yMax/3)
        y = -yMax/3 + 4*x - 3*x*x/yMax; //nasycanie
    else
        y = yMax;          //obcinanie do +yMax
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, y);
}
else
{
    right_in = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(SPI3, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, y);
}
```

Powyższy fragment programu realizuje efekt Overdrive opisany równaniem 1.12. Ponieważ wartość próbki wyjściowej nie zależy od wartości poprzednich próbek wejściowych ani wyjściowych, wartości są przechowywane w zmiennych `x`, `y`, niebędących tablicami.

#### 3.3.5.6. Tremolo

Efekt Tremolo jest realizowany w następujący sposób:

Listing 3.13. Realizacja efektu Tremolo

```
if (SPI_I2S_GetFlagStatus(I2Sx, I2S_FLAG_CHSIDE) != SET)
{
    x = SPI_I2S_ReceiveData(I2Sx);
    //modulacja amplitudy
    y = x * (1+ tremoloDepth*arm_cos_f32(2*PI*tremoloCount*tremoloFreq));
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, y);
}
else
{
    right_in = SPI_I2S_ReceiveData(I2Sx);
    while (SPI_I2S_GetFlagStatus(I2Sout, SPI_I2S_FLAG_TXE) != SET) {}
    SPI_I2S_SendData(I2Sout, y);
    tremoloCount = (tremoloCount + tremoloPeriod + 1) % (tremoloPeriod);
}
```

Efekt Tremolo jest realizowany przez modulację amplitudy sygnału wejściowego według równania 1.10. Sygnałem modulującym jest kosinusoida o częstotliwości `tremoloFreq` i głębokości modulacji `tremoloDepth`. Podobnie, jak w przypadku efektu Flanger (listing 3.10), w realizacji efektu Tremolo wykorzystano licznik `tremoloCount` o okresie `tremoloPeriod`, służący do obliczania argumentu funkcji trygonometrycznej.

### 3.3.6. Obsługa systemu przez użytkownika

Stworzony system jest przeznaczony do realizacji efektów w czasie rzeczywistym. Zasadniczym polem wykorzystania systemu jest przetwarzanie sygnału wyjściowego gitary elektrycznej. Aby uruchomić systemu, użytkownik musi włączyć zasilanie systemu poprzez wejście mini-USB układu STM32F4-Discovery. Następnie powinien podłączyć gniazdo wyjścia gitary elektrycznej do gniazda wejścia systemu (INPUT w module MMCodec01) przez kabel z monofonicznymi wtyczkami typu Jack, zaś do wyjścia układu (gniazdo typu Jack układu STM32F4-Discovery) powinien podłączyć słuchawki lub wzmacniacz audio. Zmiana realizowanego efektu następuje przez wciśnięcie przycisku „User” układu STM32F4-Discovery.

# 4. Badanie właściwości zrealizowanych efektów gitarowych

## 4.1. Test działania systemu w warunkach rzeczywistych

Zasadniczym zadaniem efektów dźwiękowych jest nadanie odpowiedniego brzmienia zmodyfikowanemu dźwiękowi. Aby ocenić poprawność działania multiefektu gitarowego i jego rzeczywistą przydatność, dokonano szeregu testów odsłuchowych systemu. W tym celu do układu podłączono gitarę elektryczną typu Les Paul (model V100 firmy Vintage [12]). Gitara ta jest wyposażona w dwa przetworniki typu Humbucker [13]. Do odsłuchiwania przetworzonego dźwięku wykorzystywano stereofoniczne słuchawki nauszne firmy Philips oraz wzmacniacz gitarowy własnej konstrukcji z wbudowanym głośnikiem (moc 20W). Odsłuchy były wykonywane w gronie znajomych osób oraz podczas seminarium Zakładu Teorii Obwodów i Sygnałów Instytutu Systemów Elektronicznych Politechniki Warszawskiej. Testy odsłuchowe wykazały poprawność realizacji wszystkich efektów, co potwierdzili zarówno uczestnicy odsłuchów, jak i sam autor.

## 4.2. Badanie wybranych efektów gitarowych

Ocena brzmienia jest miarą subiektywną. Z inżynierskiego punktu widzenia konieczna jest również bardziej obiektywna weryfikacja poprawności działania zrealizowanego systemu czasu rzeczywistego.

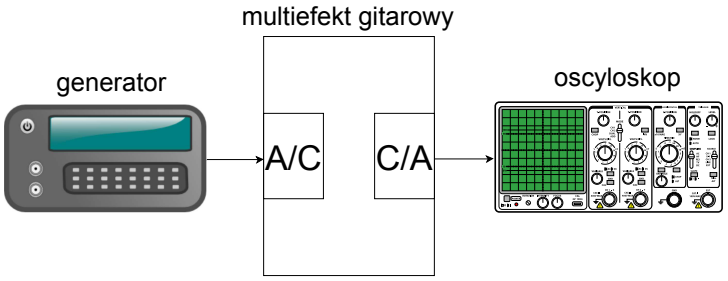
### 4.2.1. Stanowisko pomiarowe

W celu zbadania poprawności działania zrealizowanego systemu czasu rzeczywistego przeprowadzono obserwację oscyloskopową sygnału wyjściowego z przetwornika C/A systemu przy pobudzeniu wejścia przetwornika A/C sygnałem sinusoidalnym z ge-

neratora funkcyjnego. Obserwacje zostały przeprowadzone na stanowisku pomiarowym, w skład którego wchodziły:

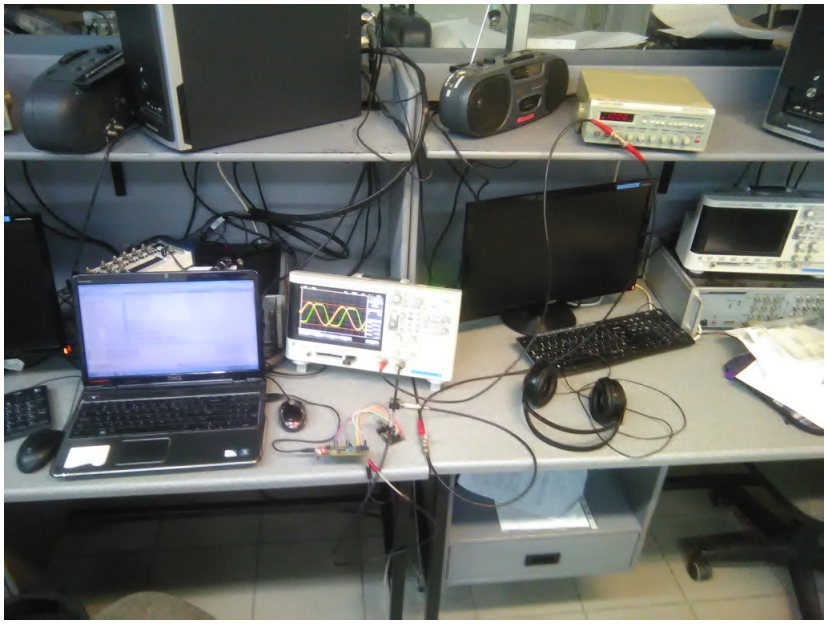
- oscyloskop cyfrowy Agilent DSO-X-2002A,
- generator funkcyjny MAXCOM MX-2020.

Poniższy rysunek przedstawia schemat układu pomiarowego:




Rys. 4.1. Schemat układu pomiarowego

Poniższa fotografia przedstawia stanowisko pomiarowe podczas badania właściwości układu:



Rys. 4.2. Stanowisko pomiarowe

### 4.2.2. Badanie działania systemu przy braku modyfikacji sygnału

Pierwszym etapem badania właściwości systemu było sprawdzenie, czy przy braku modyfikacji sygnału wejściowego, na wyjściu nie występują zniekształcenia.


Rys. 4.3. Obserwacja pracy systemu w trybie talk-through

Na powyższym oscylogramie kolorem żółtym oznaczono sinusoidalny sygnał wejściowy doprowadzony do wejścia przetwornika A/C systemu z generatora funkcyjnego, a kolorem zielonym oznaczono sygnał z wyjścia przetwornika C/A. Obserwacja nie wykazała widocznych zniekształceń sygnału. Wartość międzyszczytowa napięcia sygnału wejściowego wynosi $U_{we} = 2,5V_{pp}$, a sygnału wyjściowego $U_{wy} = 1,8V_{pp}$. Różnica poziomów sygnału wejściowego i sygnału wyjściowego wynika z faktu, że przy domyślnej konfiguracji (po zerowaniu układu) głośności przetwornika audio C/A CS43L22, maksymalna wartość międzyszczytowa sygnału wyjściowego wynosi $U_{maxC/A} = 2,05V_{pp}$ [10]. Maksymalna wartość międzyszczytowa sygnału wejściowego przetwornika A/C wynosi natomiast $U_{maxA/C} = 2,82V_{pp}$ [6]. Stosunek wartości napięcia międzyszczytowego sygnału wyjściowego do sygnału wejściowego wynosi zatem:

$$\frac{U_{wy}}{U_{we}} = \frac{U_{maxC/A}}{U_{maxA/C}} .$$

Po przekształceniu otrzymuje się:

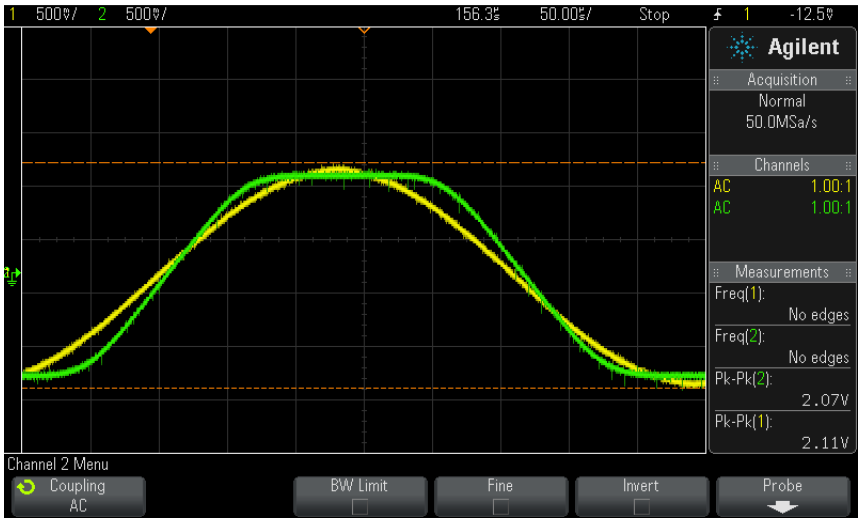
$$U_{wy} = U_{we}\frac{U_{maxC/A}}{U_{maxA/C}} = 2,5V_{pp}\frac{2,05V_{pp},}{2,82V_{pp}} = 1,8V_{pp} .$$

.

### 4.2.3. Badanie efektu Overdrive

Efekt Overdrive opisany równaniem 1.12 powinien się charakteryzować gładkim obcinaniem sinusoidy. Obserwację działania efektu Overdrive przeprowadzono dla wartości parametru $y_{max} = 1$ (*vide* równ. 1.12), co odpowiada wartości międzyszczytowej sygnału wejściowego $U_{we} = 2,82V_{pp}$ [6]. Oznacza to, że nieliniowa część

charakterystyki efektu Overdrive zaczyna się dla wartości $U_{we} = \frac{2,82V}{3} = 0,94V_{pp}$. Poniższy rysunek przedstawia oscylogram sygnału wejściowego (żółty) i wyjściowego (zielony):

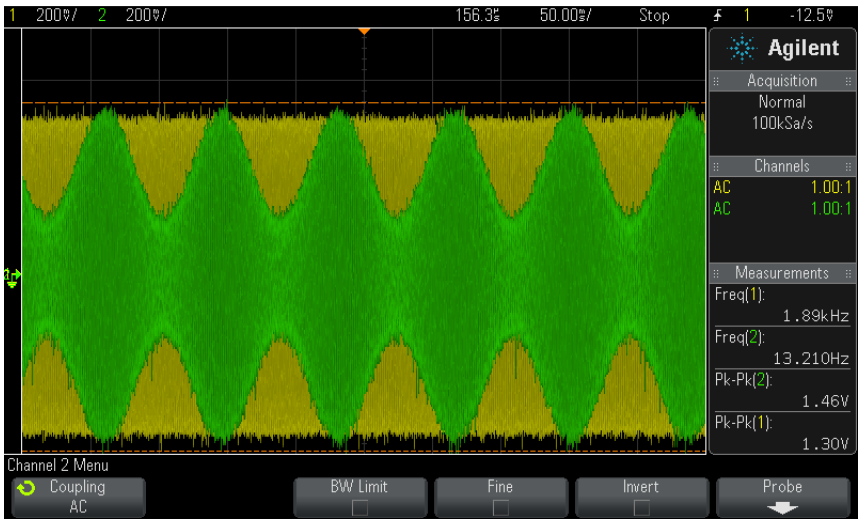



Rys. 4.4. Obserwacja działania efektu Overdrive

Można zauważyć, że sygnał wyjściowy jest przycinany do stałej wartości, ale w sygnale nie występują ostre krawędzie, co potwierdza spełnianie przez efekt stawianego mu wymagania.

### 4.2.4. Badanie efektu Tremolo

Efekt Tremolo opisany równaniem 1.10 polega na modulacji amplitudowej sygnału wejściowego. Obserwacja została przeprowadzona dla sygnału modulującego o częstotliwości $f_{mod} = 12Hz$ i amplitudzie $A = 0,5$ (*vide* równ. 1.10). Poniższy rysunek przedstawia oscylogram sygnału wejściowego (żółty) i wyjściowego (zielony):




Rys. 4.5. Obserwacja działania efektu Tremolo

Widoczne są okresowe zmiany amplitudy sygnału wyjściowego o częstotliwości odpowiadającej częstotliwości sygnału modulującego, obwiednia sygnału wyjściowego ma kształt sinusoidalny. Stosunek maksimum obwiedni do jej minimum wynosi około $\frac{1+A}{1-A} = \frac{1,5}{0,5} = 3$. Modulacja amplitudowa sygnału powoduje, że słyszalne jest „drżenie" dźwięku wyjściowego.

# 5. Podsumowanie

W ramach pracy dyplomowej zapoznano się z gitarowymi efektami dźwiękowymi oraz z algorytmami wykorzystującymi cyfrowe przetwarzanie sygnałów, pozwalającymi na ich realizację. Następnie dokonano selekcji istniejących efektów dźwiękowych w celu stworzenia funkcjonalnego zestawu popularnych efektów gitarowych. Kolejnym etapem było stworzenie aplikacji komputerowej pozwalającej na testowanie efektów dźwiękowych na sygnałach zapisanych w plikach WAV. Ostatnią częścią pracy była realizacja systemu czasu rzeczywistego służącego do przetwarzania sygnału wyjściowego gitary elektrycznej. W rezultacie powstał cyfrowy multiefekt gitarowy, którego poprawność działania została wykazana podczas testów odsłuchowych i pomiarów laboratoryjnych. Układ realizuje zestaw efektów dźwiękowych w czasie rzeczywistym, bez wprowadzania opóźnień i jest przystosowany do podłączenia gitary elektrycznej i wzmacniacza audio lub słuchawek. Dzięki temu można stosować multiefekt gitarowy do uzyskiwania efektów dźwiękowych, których realizacja byłaby możliwa przy użyciu komercyjnych urządzeń analogowych lub cyfrowych.

Zakres dziedzin, z których wiedza była potrzebna do wykonania pracy obejmuje między innymi akustykę, cyfrowe przetwarzanie sygnałów, programowanie obiektowe i programowanie mikrokontrolerów. Podczas wszystkich etapów tworzenia pracy korzystano z literatury fachowej oraz z dokumentacji technicznych.

Na podstawie wykonanej pracy można stwierdzić, że cyfrowe przetwarzanie sygnałów z pasma akustycznego jest możliwe w czasie rzeczywistym nie tylko przy użyciu wyspecjalizowanych procesorów sygnałowych, ale również mikrokontrolerów. Pozwala to na obniżenie kosztów opracowania i produkcji urządzeń służących do realizacji efektów dźwiękowych, a także stwarza możliwość prowadzenia eksperymentów dydaktycznych z dziedziny cyfrowego przetwarzania sygnałów z wykorzystaniem mikrokontrolerów ogólnego zastosowania.

# Bibliografia


[1] U. Zolzer, ed., *DAFX: Digital Audio Effects*. John Wiley & Sons, Ltd., 2002.

[2] M. Kahrs and K. Brandenburg, eds., *Applications of digital signal processing to audio and acoustics*. Kluwer Academic Publishers, 2002.

[3] S. M. Kuo, B. Lee, and W. Tian, eds., *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications*. John Wiley & Sons, 2013.

[4] STMicroelectronics, *STM32F4 Discovery Data Brief*, 2014.

[5] Propox, *MMcodec01*, 2008.

[6] Texas Instruments, *TLV320AIC23B Data Manual*, 1996.

[7] Philips Semiconductors, *AN10216-01 I2C Manual*, 2003.

[8] STMicroelectronics, *Reference manual RM0090 Rev.8*, 2014.

[9] STMicroelectronic, *STM32F405xx STM32F407xx Datasheet*, 2013.

[10] Cirrus Logic, *CS43L22*, 2010.

[11] Philips Semiconductors, *I2S bus specification*, 1996.

[12] http://www.jhs.co.uk/vintageelectric.html. (dost. 15.01.2015).

[13] pl.wikipedia.org/wiki/Humbucker. (dost. 15.01.2015).