Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Операционные системы и системное программирование (ОСиСП)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

ПРОГРАММНОЕ СРЕДСТВО ДЛЯ РАСПОЗНАВАНИЯ МУЗЫКИ

БГУИР КР 1-40 01 01 623 ПЗ

Студент: гр. 851006 Петровец В.Н.

Руководитель Жиденко А.Л.

Минск 2020

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ
Заведующий кафедрой ПОИТ
Лапицкая Н.В.

(подпись)

2020 г.

ЗАДАНИЕ
по курсовому проектированию

Студенту Петровцу Владиславу Николаевичу, 851006

1. Тема работы Программное средство для распознавания музыки
2. Срок сдачи студентом законченной работы 01.12.2020 г.
3. Исходные данные к работе Язык программирования C++/C
4. Содержание расчётно-пояснительной записки (перечень вопросов, которые подлежат разработке)

Введение.

1. Анализ прототипов, литературных источников и формирование требований к проектируемому программному средству;
2. Моделирование предметной области и разработка функциональных требований;
3. Проектирование программного средства;
4. Тестирование, проверка работоспособности и анализ полученных результатов;
5. Руководство по установке и использованию;

Список используемой литературы

Заключение

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)
1. "Программное средство для распознавания музыки", А1, схема программы, чертеж.

6. Консультант по курсовой работе
Жиденко А.Л.

7. Дата выдачи задания 05.09.2020
8. Календарный график работы над курсовой работой на весь период проектирования (с обозначением сроков выполнения и процентом от общего объёма работы):
раздел 1 к 20.09.2020 – 15 % готовности работы;
разделы 2, 3 к 13.10.2020 – 30 % готовности работы;
разделы 4, 5 к 02.11.2020 – 60 % готовности работы;
раздел 6 к 26.11.2020 – 90 % готовности работы;
оформление пояснительной записки и графического материала к 05.06.2020 – 100 % готовности работы.

РУКОВОДИТЕЛЬ_____Жиденко А.Л.
(подпись)

Задание принял к исполнению Петровец В.Н._____.
(дата и подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Представьте себе ситуацию: вы сидите в кафе или едете в такси и тут по радио начинает играть какая-либо песня, которая так сильно западает в душу, что вы готовы крутить ее хоть всю неделю. Проблема в том, что в такси не могут ответить вам, что за волшебный трек вы сейчас слышали. Конечно, будет легче, если мелодия была на английском. Вы можете просто забить в поисковике слова из песни, а вот если трек звучал на испанском или французском, (а именно с этими языками у вас в школе были сложности), то найти его будет куда сложней.

Сколько раз вы слышали понравившуюся песню по радио или где-то на улице и хотели узнать ее название и исполнителя, чтобы затем скачать и слушать у себя? Это происходит с каждым из нас время от времени. К счастью, современные технологии позволяют искать мелодии по звуку, сравнивать их с базой данных и показывать названия и исполнителя прямо на экране телефона.

Все это возможно благодаря высоким достижениям человечества в областях программирования и математического анализа. Была поставлена задача изучить теорию, необходимую для полного понимания и анализа проблемы.

Необходимо понимать, что среди всех характеристик звука, неизменной остается частотно-амплитудная характеристика, которую и нужно было получить для успешной реализации алгоритма. Для ее получения есть несколько математических автоматов, описание которых можно найти в статьях на просторах интернета.

1 АНАЛИЗ ПРОТОТИПОВ, ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ И ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К ПРОЕКТИРУЕМОМУ ПРОГРАММНОМУ СРЕДСТВУ

1.1 Анализ существующих аналогов

В ходе анализа существующих средств для распознавания песен было выяснено, что десктопных приложений для решения данной задачи попросту не существует. Однако, стоит отметить, что существуют аналоги проектируемого программного средства на платформе Android и iOS. Все приложения используют микрофон для записи звука и имеют схожий функционал.

1.1.1 Мобильное приложение “Shazam”

Возглавляет список аналогов мобильное приложение Shazam, разработанное компанией Apple. Shazam – бесплатный кроссплатформенный проект, позволяющий пользователю определить, что за песня играет в данный момент времени. Интерфейс приведен на рисунке 1.

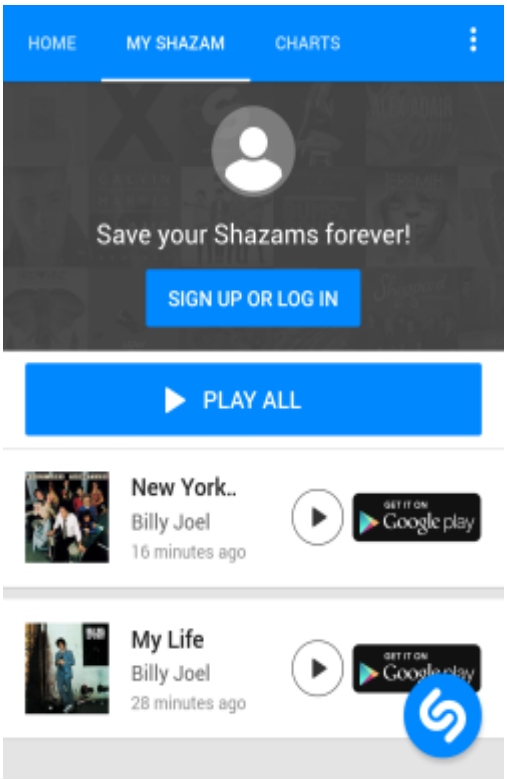


Рисунок 1 — Интерфейс приложения “Shazam”

Приложение Shazam имеет свои преимущества и недостатки.

Недостатки:

– существует только в мобильной версии.

Преимущества:

– высокая скорость и точность поиска песен;

– огромная база песен – более 11 млн треков;

– успешная работа даже при сильных помехах в виде шума;

– предоставление почти всей информации о композиции после поиска;

– интегрирован с Facebook, Twitter, Google+, Whatsapp и другими, что существенно облегчает обмен идентифицированными композициями с друзьями;

– сохранение истории поиска;

– возможность использования off-line с помощью сохранения записанного сэмпла и последующего поиска при появлении сети.

1.1.2 Мобильное приложение “SoundHound”

Бесспорно, главным конкурентом Shazam можно считать бесплатное приложение SoundHound. Как и в случае с Shazam, после поиска возможно узнать не только название песни, но и множество другой информации, включая видеоклип и текст песни. По словам разработчиков, приложение может распознать даже напетую пользователем песню. Интерфейс приведен на рисунке 2.

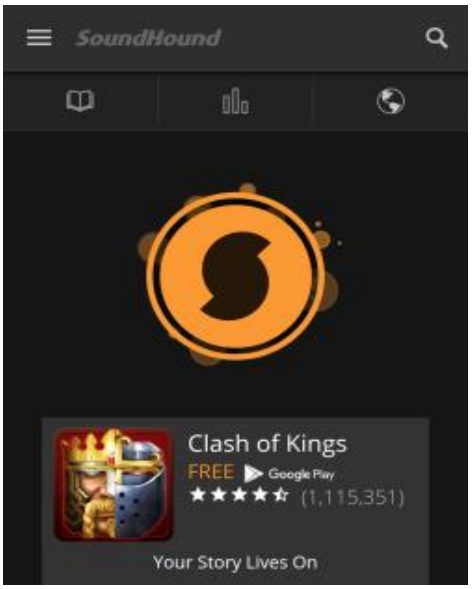


Рисунок 2 — Интерфейс приложения “SoundHound”

Недостатки мобильного приложения “SoundHound”:

– нет возможности работать с приложением off-line;

– скорость распознавания песни чуть ниже, чем у Shazam.

Преимущества приложения:

– высокая точность поиска песен;

– возможность следить за текстом песни в реальном времени;

– предоставление почти всей информации о композиции после поиска.

1.1.3 Мобильное приложение “BeatFind”

По точности распознавания почти не уступает своим старшим братьям. Одним из главных плюсов приложения является тот факт, что BeatFind практически лишён рекламы и прочих нагромождающих вещей в виде навороченного интерфейса. Интерфейс приведен на рисунке 3.

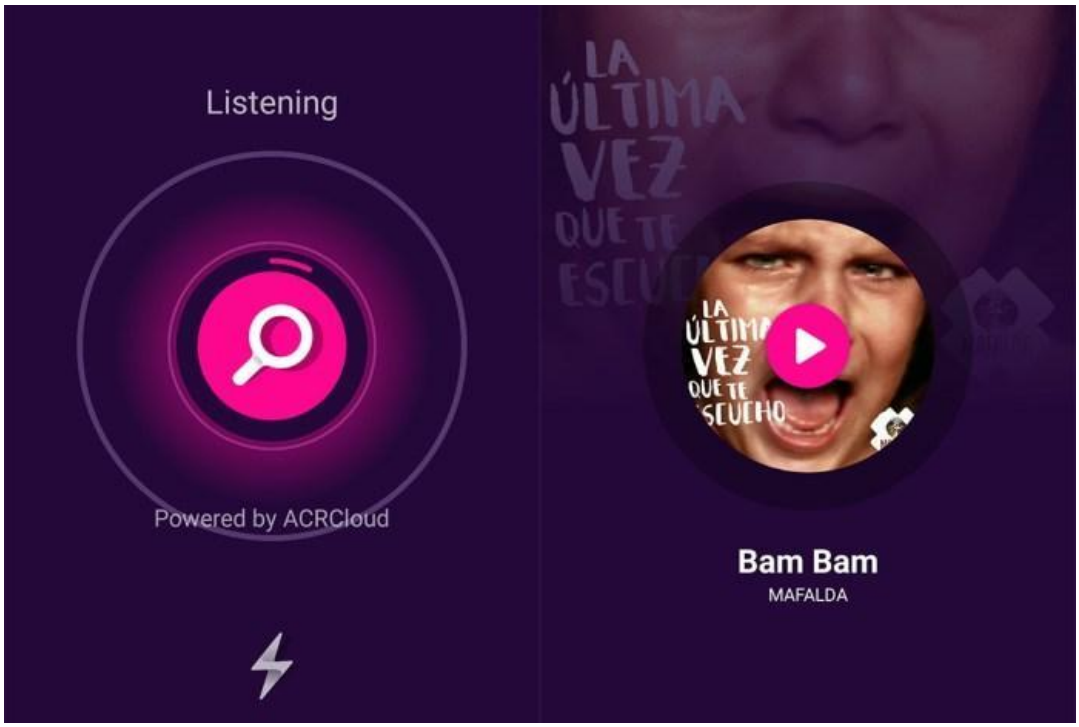


Рисунок 3 — Интерфейс приложения “BeatFind”

Недостатки мобильного приложения “BeatFind”:

– нет возможности работать с приложением offline, то есть без соединения с интернетом;

– сравнительно низкая скорость распознавания.

Преимущества приложения:

– высокая точность поиска песен;

– приятный бонус в виде визуализации музыки, которая играет.

1.1.4 Мобильное приложение “MusicID”

Ещё один малоизвестный сервис с минимумом рекламы внутри. В большей степени, приложение MusicID ориентировано на зарубежную публику, поэтому отечественные композиции даются ему с трудом. Опознанный трек дополняется ссылками на Amazon.

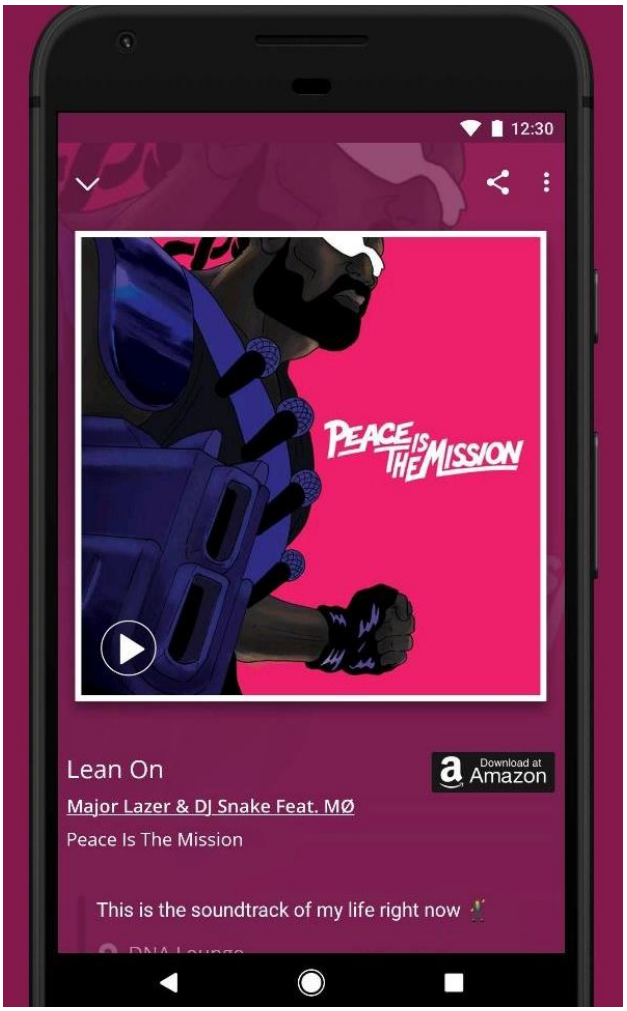


Рисунок 4 — Интерфейс приложения “MusicID”

Недостатки мобильного приложения “MusicID”:

– нет возможности работать с приложением offline, то есть без соединения с интернетом;

– сравнительно низкая скорость распознавания;

– не поддерживается русский язык.

Преимущества приложения:

– высокая точность поиска песен;

– составление списка рекомендаций в связи с историей поиска;

– предоставление ссылок на данный трек в посторонние сервисы, такие как Deezer или YouTube.

1.2 Постановка задачи

В задачу курсового проекта входит создание клиент-серверного приложения на языке C++ в интегрированной среде разработки C++ Builder, которое реализует поиск аудиозаписи по записанному с микрофона участка этой аудиозаписи. Необходимо предусмотреть возможность сохранения и проигрывания записанного участка, а также пополнения базы песен владельцем сервера.

2 МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ И РАЗРАБОТКА ФУНКЦИОНАЛЬНЫХ ТРЕБОВАНИЙ

2.1 Описание алгоритма получения записи с микрофона

Ввиду того, что встроенные процедуры для работы со звуком не удовлетворяли потребностей разработчика, было решено использовать библиотеку “bass.lib”.

2.1.1 Запись звука с микрофона

Библиотека “bass.lib” имеет широкий спектр инструментов для работы со звуком как угодно программисту. Сначала необходимо настроить свойства канала ввода: частоту дискретизации звука, глубину звука, количество каналов, формат модуляции, смещение блоков и среднее количество байтов в секунду. Обо всем по порядку.

Немного теории. Аналоговый звук – это непосредственно то, что человек слышит ушами, т.е. колебания воздуха в пространстве. Для того, чтобы предоставить компьютеру возможность работать со звуком, его нужно представить в цифровом виде, т.е. оцифровать. Цифровой звук – это набор координат, описывающих звуковую волну. Оцифровка звука включает в себя 3 этапа: дискретизация, квантование и кодирование. Ниже приведен график зависимости некоторой волны f(t) от времени t (см. рисунок 5).

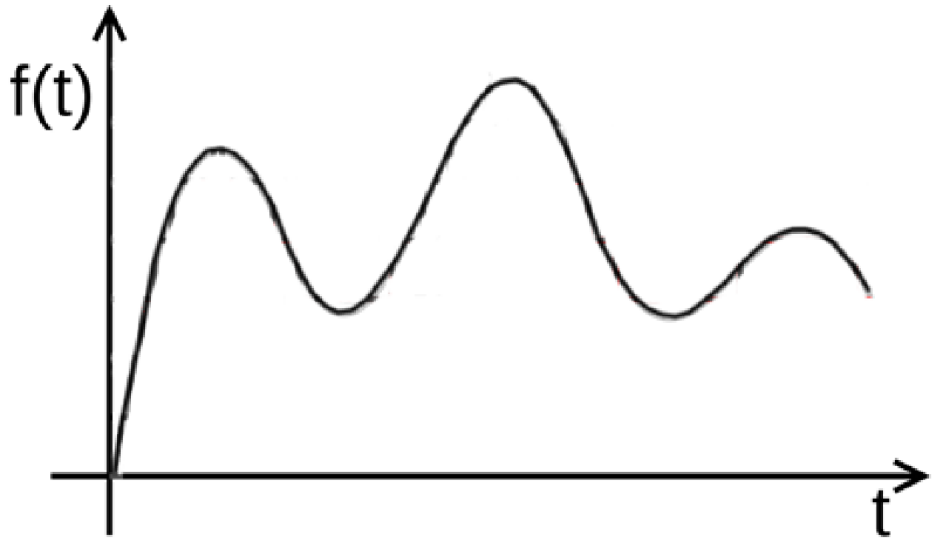


Рисунок 5 — График зависимости некоторой волны f(t)

Дискретизация – представление аналогового непрерывного сигнала в виде выборки его значений через определенные промежутки времени. Значения сигнала в одной выборке считается постоянным. Квантование – разбиение сигнала на конечное число уровней и округление этих значений до ближайшего из двух уровней. Дискретизация – разбиение функции по оси времени, квантование – по оси значений. Пример – на рисунках 6, 7 и 8.

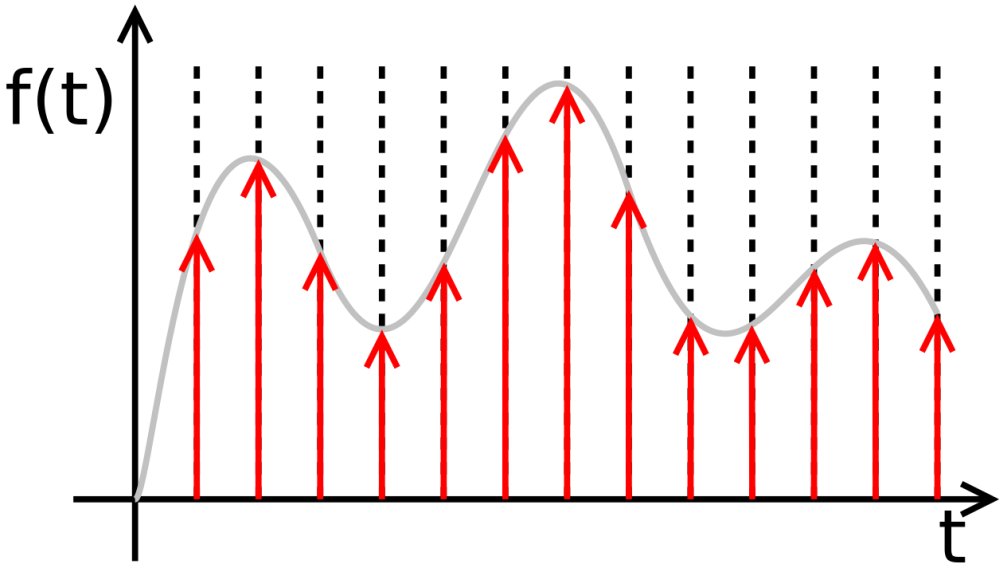


Рисунок 6 — Дискретизация сигнала f(t)

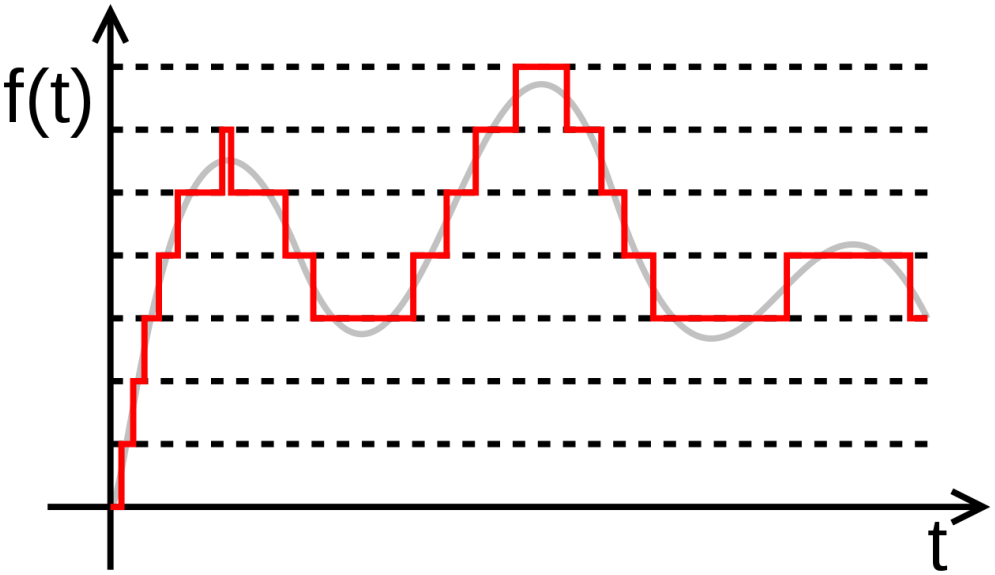


Рисунок 7 — Квантование сигнала f(t)

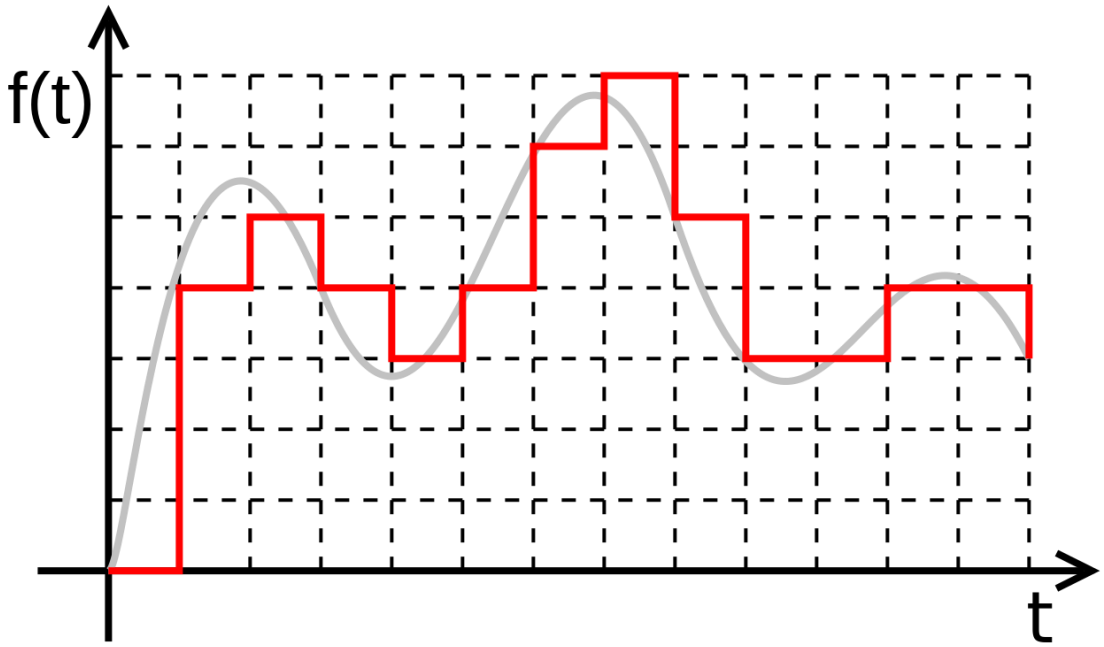


Рисунок 8 — Цифровой сигнал f(t)

Теперь следует пояснить термины, приведенные при описании свойств канала. Частота дискретизации – количество выборок сигнала в секунду. Глубина звука (глубина квантования, разрядность) – количество битов, необходимое для кодирование всех уровней сигнала. Для случая, изображенного на графике: количество уровней равно 8, значит глубина квантования равна 3 ($8 = 2^3$). Количество каналов равно одному для монофонического звука, двум для стереофонического звука. Формат модуляции – PCM (импульсно-кодовая модуляция). Смещение блока равно количеству каналов на глубину звука в байтах. Среднее количество байтов в секунду – произведение частоты дискретизации на смещение блока.

После определения свойств канала необходимо указать устройство, с которого будет осуществляться ввод звука, выбрать поток ввода с этого устройства и реализовать CALLBACK процедуру для корректной работы записи.

2.1.2 Сохранение в файл

Запись оцифрованного звука велась в настроенный ранее канал. Следовательно, канал – массив байтов. Записать массив байтов в бинарный файл – задача несложная. Для более простой реализации алгоритма было решено работать с .wav файлами. Выбор пал на этот формат по двум причинам: это один из немногих форматов, которые не подвергаются сжатию;

его структура довольно проста – заголовок + данные, поэтому считать оцифрованный звук из файла очень легко.

2.1.3 Воспроизведение записанного звука

Для воспроизведения звука из файла можно было воспользоваться встроенными библиотеками, но, раз уже подключена сторонняя библиотека, было решено подробнее разобраться с потоками вывода. В отличие от встроенных процедур, в подключенных была возможность также настроить канал вывода. Алгоритм схож с записью: настройка канала, выбор воспроизводящего устройства (динамик), выбор потока вывода, непосредственно воспроизведение.

2.2 Описание алгоритма распознавания

Необходимо понимать, что у двух одинаковых песен, записанных в разной обстановке, останется одинаковым частотно-амплитудная характеристика. Для ее получения можно воспользоваться таким замечательный математическим аппаратом, как быстрое преобразование Фурье (далее БПФ). Входными данными для БПФ являлся массив комплексных чисел, действительная часть которых – образцы цифрового сигнала, мнимая – ноль. Результатом БПФ является спектр – комплексная матрица, представляющая собой список коэффициентов комбинаций комплексных синусоид, упорядоченных по частоте. Значения спектра – комплексные числа, модуль которых – амплитуда данной частоты, аргумент – соответствующая начальная фаза (иначе говоря, результат БПФ – амплитудный и фазовый спектр). Из спектра выбираются частоты с максимальными амплитудами в некоторых диапазонах. Эти частоты формируют цифровую сигнатуру для участка песни. Сигнатуры используются в качестве ключей для создания хеш-таблицы. Таким образом, хеш-таблица записанного на микрофон участка сравнивается с хеш-таблицей песен из базы. Чем больше совпадений, тем выше вероятность, что записанный участок принадлежит этой песне.

2.3 Описание функциональности программного средства

Интерфейс программного средства должен иметь простой вид: 5 кнопок (начать запись, остановить запись, распознать запись, воспроизвести запись,

добавить песню в базу данных) и поле для вывода результатов. Результат программы – список песен с количеством совпадений сигнатур для каждой.

При нажатии на кнопку “Начать запись” должна начаться запись звука, причем эта кнопка должна быть недоступна для повторного использования во избежание уязвимостей (как и кнопки “Распознать” “Воспроизвести”). При нажатии на кнопку “Остановить запись” процесс записи должен остановиться, и запись должна сохраниться в буферный файл. После этого появляется доступ к кнопкам “Распознать ” и “Проиграть”, что свидетельствует о корректном сохранении файла и корректной записи дорожки. При нажатии на кнопку “Воспроизвести” запись должна воспроизвестись. В случае, если пользователя не удовлетворяет данная запись, он должен иметь возможность перезаписать её.

3 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

3.1 Разработка используемого аудиоформата

Помимо отсутствия сжатия, wav формат имеет очень простую структуру. Он представляет собой две чётко разграниченные области. Одна из них – заголовок файла, другая – область данных. В заголовке хранится информация о размере файла, количестве каналов, частоте дискретизации и глубине звучания, а также некоторая информация об исполнителе, годе выпуска и длине записи в секундах.

Размер заголовка строго фиксирован – 44 байта. К примеру, у mp3 файла размер заголовка хранится непосредственно в первых байтах заголовка. Кроме того, область данных mp3 файла разделена на фреймы, состоящие из заголовка фрейма и некоторого фиксированного размера данных. Как можно было понять, mp3 файл – не лучший вариант для программного обслуживания. Следует упомянуть, что mp3 формат подразумевает сжатие звука, что ведет к потере точной информации о звуке.

Давайте рассмотрим заголовок .wav файла подробнее в таблице 3.1. Как видно из таблицы, после 44 байт наступает область данных.

Таблица 3.1 – Заголовок .wav формата

Местоположение	**Поле**	**Описание**
0…3 (4 байта)	chunkId	Содержит символы «RIFF» в ASCII кодировке 0x52494646. Является началом RIFF-цепочки.
4…7 (4 байта)	chunkSize	Это оставшийся размер цепочки, начиная с этой позиции. Иначе говоря, это размер файла минус 8, то есть, исключены поля chunkId и chunkSize.
8…11 (4 байта)	format	Содержит символы «WAVE» 0x57415645

Продолжение таблицы 3.1

Местоположение	**Поле**	**Описание**
12…15 (4 байта)	subchunk1Id	Содержит символы "fmt " 0x666d7420
16…19 (4 байта)	subchunk1Size	16 для формата PCM. Это оставшийся размер подцепочки, начиная с этой позиции.
20…21 (2 байта)	audioFormat	Аудио формат, список допустипых форматов. Для PCM = 1 (то есть, Линейное квантование). Значения, отличающиеся от 1, обозначают некоторый формат сжатия.
22…23 (2 байта)	numChannels	Количество каналов. Моно = 1, Стерео = 2 и т.д.
24…27 (4 байта)	sampleRate	Частота дискретизации. 8000 Гц, 44100 Гц и т.д.
28…31 (4 байта)	byteRate	Количество байт, переданных за секунду воспроизведения.
32…33 (2 байта)	blockAlign	Количество байт для одного сэмпла, включая все каналы.
34…35 (2 байта)	bitsPerSample	Количество бит в сэмпле. Так называемая «глубина» или точность звучания. 8 бит, 16 бит и т.д.
36…39 (4 байта)	subchunk2Id	Содержит символы «data» 0x64617461
40…43 (4 байта)	subchunk2Size	Количество байт в области данных.

Продолжение таблицы 3.1

Местоположение	Поле	Описание
44…	data	Непосредственно WAV-данные.

3.2 Разработка алгоритма распознавания

Данные из временной области (образцы звука) помещаются в действительные поля комплексной матрицы с мнимыми полями, равными нулю. Для реализации прямоугольного окна (англ. Chunk) вводились такие параметры, как ChunkSize и SampledChunkCount. ChunkSize – размер окна в байтах, эквивалентный звучанию примерно одной секунды аудио. SampledChunkCount – количество этих окон во всей записи. После заполнения исходной комплексной матрицы complexArray, содержащей информацию об одном окне, запускается БПФ-анализ для этого окна. Результат БПФ-анализа всех окон – двумерная комплексная матрица, столбцы которой – соответствующие частоты, а строки – соответствующее окно. Значения на пересечении столбцов и строк – комплексные числа, действительная часть которых – коэффициент амплитуды, мнимая – начальная фаза. Таким образом, совокупность всех действительных частей составляет амплитудный спектр, мнимых – фазовый спектр. Вместе они составляют суммарный спектр произведения, т.е. комплексную функцию, описывающую гармоники песни. Для распознавания записанной звуковой дорожки сравнивались цифровые подписи песен, находящихся в базе данных и цифровая подпись записанной звуковой дорожки. Чем больше совпадений – тем больше вероятность, что данная песня – искомая.

Для реализации БПФ был использован модуль KISS_FFT, найденный на просторах интернета в открытом доступе.

3.3 Разработка алгоритма создания цифровой подписи песни

Для создания цифровой сигнатуры произведения необходимо выбрать из заданных ранее диапазонов самые важные частоты, т.е. частоты с максимальной амплитудой. Для этого необходимо ввести несколько диапазонов частот, в пределах которых выбиралась самая громкая частота. Так как частоты музыкальных инструментов бывают не выше 300 Гц, было решено выбрать следующие диапазоны: 1-40 Гц, 41-80 Гц, 81-120 Гц, 121-180 Гц, 180-300 Гц. Самые громкие частоты формируют хеш-значения для каждого окна

произведения. Хеш-функция представляла эти значения в виде большого числа типа long. Совокупность всех чисел представляла собой цифровую сигнатуру произведения. Эта совокупность записывалась в соответствующую хеш-таблицу. Подсчет совпадений ведётся при вычислении хеш-значений для каждого окна записанного звука. На основании количества совпадений делается вывод о принадлежности записанного участка какой-либо песне.

На вход хеш-функции подавались 4 значения самых громких частот из введенных интервалов. Хеш-функция представляла эти 4 числа в виде десятизначного десятеричного числа: три цифры на диапазон 121-180 Гц, три цифры на диапазон 81-120 Гц, две цифры на диапазон 41-80 Гц, две цифры на диапазон 1-40 Гц. Кроме этого, было необходимо округлять частоты до ближайшего четного числа в меньшую сторону для учета погрешности вычисления частот.

Например, на вход были поданы значения 37, 58, 93, 130. После округления получились частоты 36, 58, 92, 130. Полученное хеш-значение – 1300925836.

Общая схема алгоритма приведена в приложении 1.

4 ТЕСТИРОВАНИЕ, ПРОВЕРКА РАБОТОСПОСОБНОСТИ И АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ

Тестирование программного средства производилась на персональном компьютере с установленной операционной системой Windows 10. В качестве записывающего устройства были использованы встроенный микрофон и микрофон бренда Defender.

4.1 Тестирование функционала программы

Таблица 4.1 — Тестирование функционала программы

Номер теста	**Тестируемая функциональность**	**Последовательность действий**	**Ожидаемый результат**
1	Запись и воспроизведение звука со встроенного микрофона.	1. Нажать кнопку Start Recording; 2. Сказать в микрофон кодовую фразу; 3. Нажать кнопку Stop Recording; 4. Нажать кнопку Play.	Воспроизведение кодовой фразы
2	Запись и воспроизведение звука с подключенного микрофона.	1. Нажать кнопку Start Recording; 2. Сказать в микрофон кодовую фразу; 3. Нажать кнопку Stop Recording; 4. Нажать кнопку Play.	Воспроизведение кодовой фразы
3	Распознавание песни, записанной близко к микрофону с отсутствием шума.	1. Нажать кнопку Start Recording; 2. Проиграть в микрофон отрывок песни; 3. Нажать кнопку Stop Record; 4. Нажать кнопку Recognise.	Однозначное распознавание песни

Продолжение таблицы 4.1

Номер теста	**Тестируемая функциональность**	**Последовательность действий**	**Ожидаемый результат**
4	Распознавание песни, записанной близко к микрофону с присутствием шума.	1. Нажать кнопку Start Recording; 2. Проиграть в микрофон отрывок песни и подпевать; 3. Нажать кнопку Stop Recording; 4. Нажать кнопку Recognise.	Однозначное распознавание песни
5	Распознавание песни, записанной далеко от микрофона с отсутствием шума.	1. Нажать кнопку Start Recording; 2. Проиграть в микрофон отрывок песни; 3. Нажать кнопку Stop Recording; 4. Нажать кнопку Recognise.	Однозначное распознавание песни
6	Распознавание песни, записанной далеко от микрофона с присутствием шума.	1. Нажать кнопку Start Record; 2. Проиграть в микрофон отрывок песни и подпевать; 3. Нажать кнопку Stop Record; 4. Нажать кнопку Recognise.	Неоднозначное распознавание песни

4.2 Вывод из прохождения тестирования

Программа успешно прошла все тесты, что показывает корректность работы программы и соответствие функциональным требованиям.

Подключение микрофона происходило без проблем, как и работа среды с ним. В большинстве случаев, для корректного распознавания песни хватало десяти секунд записи.

5 РУКОВОДСТВО ПО УСТАНОВКЕ И ИСПОЛЬЗОВАНИЮ

Для того, чтобы приступить непосредственно к распознаванию песен необходимо запустить сервер и клиент. Дополнительных установок не требуется, рекомендуется занести все компоненты, составляющие программу, в отдельную папку. Интерфейс разработанного средства представлен на рисунке 10.

Рисунок 10 — Интерфейс приложения

Для начала и остановки записи предназначены кнопки Start Recording и Stop Recording соответственно.

После записи пользователь может прослушать записанный участок (нажав на кнопку Replay) или запустить процесс поиска (нажав на кнопку Recognize).

Рисунок 12 — Процесс записи

Если после нажатия кнопки Start recording появилась ошибка “Record device doesn't inizialized”, следует проверить драйвера подключенного устройства.

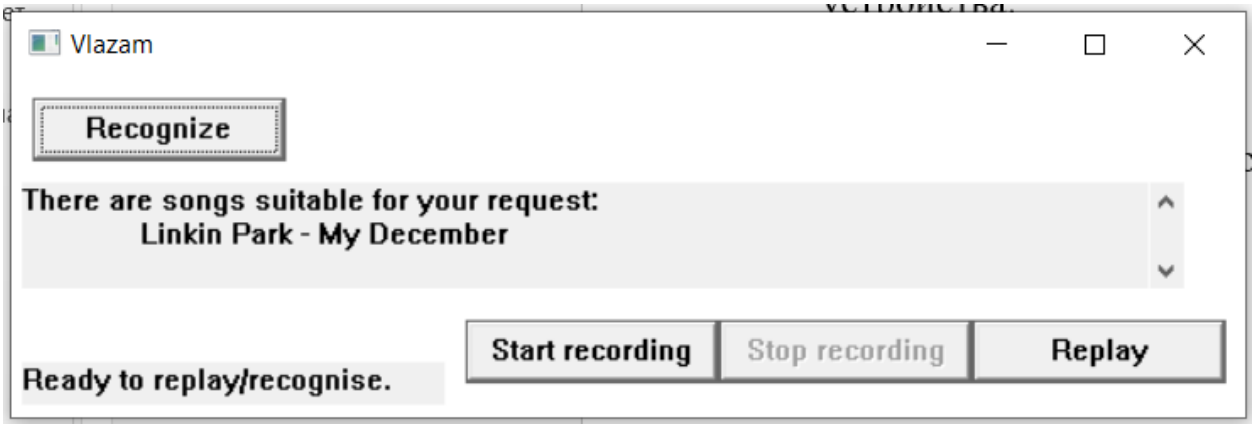


Рисунок 13 — Результат работы клиента

ЗАКЛЮЧЕНИЕ

В рамках данного курсового проекта был произведен анализ предметной области и реализовано программное и информационное обеспечение "Программное средство для распознавания музыки". Согласно поставленным задачам в данном приложении были реализованы такие возможности, как распознавание музыки по записи с микрофона, сохранение записанного аудиофайла, пополнение базы данных программы, отправка данных на сервер и получение данных с сервера.

Спроектированная система имеет интуитивно понятный интерфейс для пользователя, проста в использовании, не требует серьезных аппаратных затрат.

Для успешного выполнения всех поставленных задач потребовалось изучить основы WINAPI и изучить документацию по библиотеке “bass.dll”.

Разработанное программное средство, на удивление, работает очень хорошо. Среди примерно десяти тестов только один давал неправильный результат, но разбежка между полученным и правильным результатом составляла не более пяти совпадений.

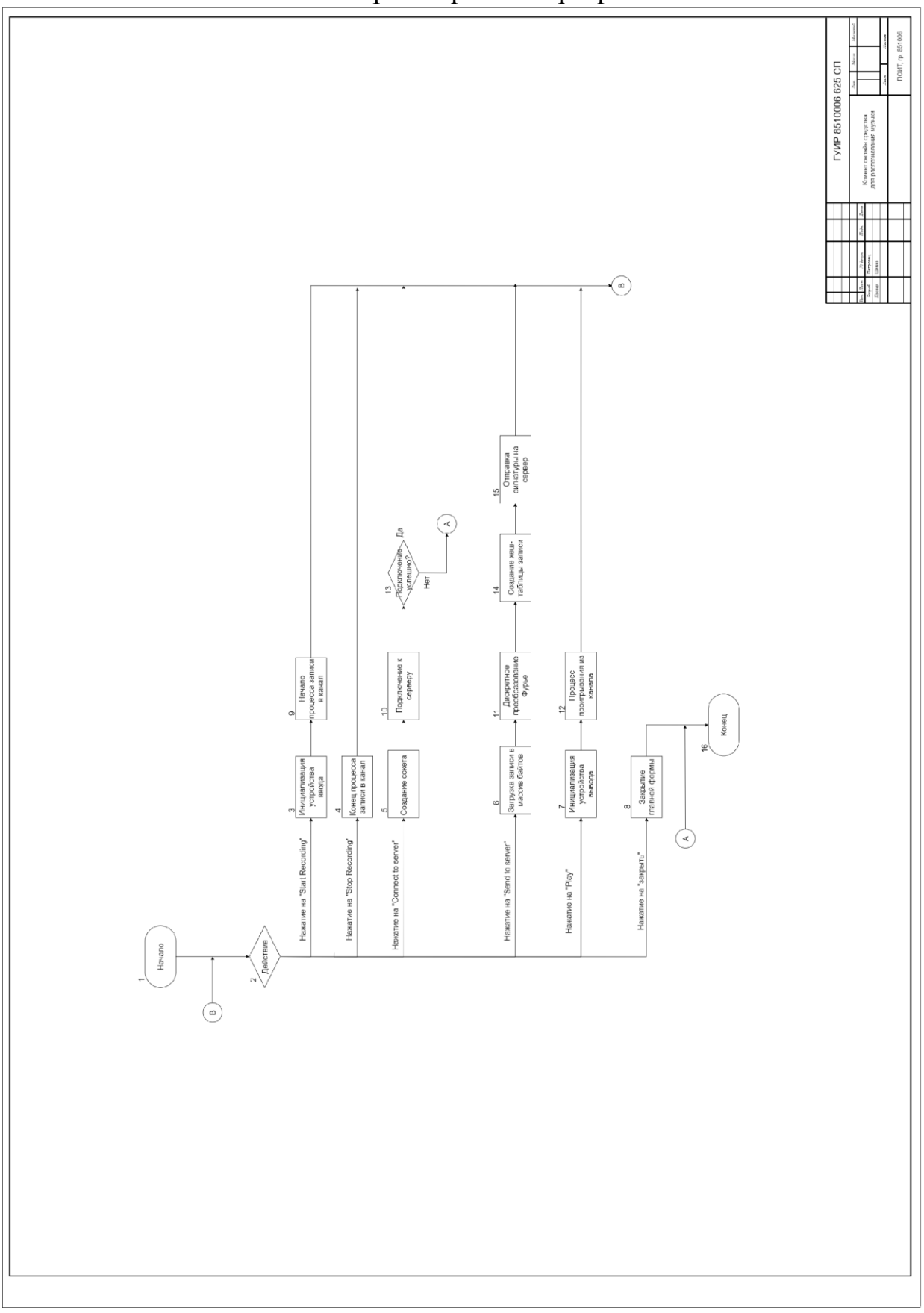
На ряду с высокой точностью работы, стоит оценить время, затраченное на распознавание трека. После пяти секунд проигрывания, цикл распознавания занимает не более трех секунд, а точность составляет до семидесяти процентов. После десяти секунд проигрывания, цикл распознавания занимает до пяти секунд, а точность составляет около девяноста процентов. Оптимальное время, за которое программа сможет распознать песню – от семи до двенадцати секунд.

СПИСОК ЛИТЕРАТУРЫ

[1] Wikipedia [Электронный ресурс] Режим доступа: https://ru.wikipedia.org/wiki/ — Дата доступа: 01.06.2020.
[2] BASS.DLL Documentation [Электронный ресурс] Режим доступа: http://www.un4seen.com/doc/#bass/DSPPROC.html — Дата доступа: 02.06.2020.
[3] Центр разработки для Windows [Электронный ресурс] Режим доступа: https://docs.microsoft.com/ru-ru/windows/win32/api/ — Дата доступа: 03.06.2020.
[4] Crossplatform.ru [Электронный ресурс] Режим доступа: http://doc.crossplatform.ru/ — Дата доступа: 04.06.2020.
[5] github.com [Электронный ресурс] Режим доступа: https://github.com/mborgerding/kissfft — Дата доступа: 05.06.2020.

ПРИЛОЖЕНИЕ А

Схема алгоритма работы программы



ПРИЛОЖЕНИЕ Б

Исходный код программы

Source.cpp

```
#include <Windows.h>
#include <windowsx.h>
#include "Vlazam.h"

#define STATIC_RESULTS_INTRO "There are songs suitable for your request:"

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
int CALLBACK wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR szCmdLine, int
nCmdShow);
LRESULT CALLBACK BtnStartRecordingClick();
LRESULT CALLBACK BtnStopRecordingClick();
LRESULT CALLBACK BtnReplayClick();
LRESULT CALLBACK BtnRecognizeClick();

static HWND hBtnStartRecording, hBtnStopRecording, hBtnReplay, hBtnRecognize,
hStaticStatus, hStaticResults;

LRESULT CALLBACK BtnStartRecordingClick() {
    if (startRecording() == -1) {
        return EXIT_FAILURE;
    }

    Static_SetText(hStaticStatus, "Recording...");

    Button_Enable(hBtnStartRecording, FALSE);
    Button_Enable(hBtnStopRecording, TRUE);

    Button_Enable(hBtnRecognize, FALSE);
    Button_Enable(hBtnReplay, FALSE);

    return EXIT_SUCCESS;
}

LRESULT CALLBACK BtnStopRecordingClick() {
    if (stopRecording() == -1) {
        return EXIT_FAILURE;
    }
    if (saveRecording(RECORDED_BUF_FILENAME) == -1) {
        return EXIT_FAILURE;
    }

    Static_SetText(hStaticStatus, "Ready to replay/recognise.");

    Button_Enable(hBtnStartRecording, TRUE);
    Button_Enable(hBtnStopRecording, FALSE);
    Button_Enable(hBtnRecognize, TRUE);
    Button_Enable(hBtnReplay, TRUE);

    return EXIT_SUCCESS;
}

LRESULT CALLBACK BtnReplayClick() {
    int err = 0;
    if ((err = playFileWAV(RECORDED_BUF_FILENAME)) != 0) {
        return err;
    }
```

```
    Static_SetText(hStaticStatus, "Replaying...");
    Button_Enable(hBtnStartRecording, FALSE);
    Button_Enable(hBtnStopRecording, FALSE);
    Button_Enable(hBtnRecognize, FALSE);
    Button_Enable(hBtnReplay, FALSE);
    waitTillPlaying();
    Button_Enable(hBtnStartRecording, TRUE);
    Button_Enable(hBtnStopRecording, FALSE);
    Button_Enable(hBtnRecognize, TRUE);
    Button_Enable(hBtnReplay, TRUE);
    Static_SetText(hStaticStatus, "Ready to replay/recognise.");

    return EXIT_SUCCESS;
}

LRESULT CALLBACK BtnRecognizeClick() {
    int num, res;
    char** results = nullptr;

    Static_SetText(hStaticStatus, "Recognizing...");

    if (recognizeSample(results, num) == -1) {
        Static_SetText(hStaticStatus, "Some errors occur.");
        return EXIT_FAILURE;
    }

    int staticSumLen = 0;
    for (int i = 0; i < num; i++) {
        staticSumLen += strlen(results[i]) + 2;
    }
    staticSumLen += strlen(STATIC_RESULTS_INTRO) + 1;
    char* staticResultsText = (char*)malloc(sizeof(char) * staticSumLen);
    if (!staticResultsText) {
        return EXIT_FAILURE;
    }
    memset(staticResultsText, 0, staticSumLen);
    strcpy_s(staticResultsText, staticSumLen, STATIC_RESULTS_INTRO);
    for (int i = 0; i < num; i++) {
        strcat_s(staticResultsText, staticSumLen, "\n\t");
        strcat_s(staticResultsText, staticSumLen, results[i]);
    }

    Static_SetText(hStaticResults, staticResultsText);

    Static_SetText(hStaticStatus, "Ready to replay/recognise.");

    return EXIT_SUCCESS;
}

int CALLBACK wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR szCmdLine, int
nCmdShow) {
    MSG msg{};
    HWND hWnd{};
    WNDCLASSEX wc{ sizeof(WNDCLASSEX) };

    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hbrBackground = reinterpret_cast<HBRUSH>(GetStockObject(WHITE_BRUSH));
    wc.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wc.hIcon = LoadIcon(nullptr, IDI_APPLICATION);
    wc.hIconSm = LoadIcon(nullptr, IDI_APPLICATION);
    wc.hInstance = hInstance;
```

```
    wc.lpfnWndProc = WndProc;
    wc.lpszClassName = TEXT("MyAppClass");
    wc.lpszMenuName = nullptr;
    wc.style = CS_VREDRAW | CS_HREDRAW; //!!!

    if (!RegisterClassEx(&wc))
        return EXIT_FAILURE;

    if ((hWnd = CreateWindow(wc.lpszClassName, TEXT("Vlazam"),
                WS_OVERLAPPEDWINDOW, 0, 0, 600, 200, nullptr, nullptr,
                wc.hInstance, nullptr)) == INVALID_HANDLE_VALUE)
        return EXIT_FAILURE;

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    while (GetMessage(&msg, nullptr, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    int err = 0;
    HINSTANCE hInst;

    switch (uMsg) {
    case WM_CREATE:
        hInst = ((LPCREATESTRUCT)lParam)->hInstance;

        hBtnStartRecording = CreateWindow("button", "Start recording",
            WS_CHILD | WS_VISIBLE | WS_BORDER,
            215, 115, 120, 30, hWnd, 0, hInst, NULL);
        ShowWindow(hBtnStartRecording, SW_SHOWNORMAL);
        UpdateWindow(hBtnStartRecording);

        hBtnStopRecording = CreateWindow("button", "Stop recording",
            WS_CHILD | WS_VISIBLE | WS_BORDER,
            335, 115, 120, 30, hWnd, 0, hInst, NULL);
        ShowWindow(hBtnStopRecording, SW_SHOWNORMAL);
        UpdateWindow(hBtnStopRecording);
        Button_Enable(hBtnStopRecording, FALSE);

        hBtnReplay = CreateWindow("button", "Replay",
            WS_CHILD | WS_VISIBLE | WS_BORDER,
            455, 115, 120, 30, hWnd, 0, hInst, NULL);
        ShowWindow(hBtnReplay, SW_SHOWNORMAL);
        UpdateWindow(hBtnReplay);
        Button_Enable(hBtnReplay, FALSE);

        hBtnRecognize = CreateWindow("button", "Recognize",
            WS_CHILD | WS_VISIBLE | WS_BORDER,
            10, 10, 120, 30, hWnd, 0, hInst, NULL);
        ShowWindow(hBtnRecognize, SW_SHOWNORMAL);
        UpdateWindow(hBtnRecognize);
        Button_Enable(hBtnRecognize, FALSE);

        hStaticStatus = CreateWindow("static", "Status",
            WS_CHILD | SS_LEFTNOWORDWRAP,
            5, 135, 200, 20, hWnd, 0, hInst, NULL);
```

```
        ShowWindow(hStaticStatus, SW_SHOWNORMAL);
        UpdateWindow(hStaticStatus);
        Static_SetText(hStaticStatus, "");

        hStaticResults = CreateWindow("static", "",
            WS_CHILD | SS_EDITCONTROL | WS_VSCROLL,
            5, 50, 550, 50, hWnd, 0, hInst, NULL);
        ShowWindow(hStaticResults, SW_SHOWNORMAL);
        UpdateWindow(hStaticResults);

        err = BassDllInit();
        if (err != 0) {
            MessageBox(hWnd, TEXT("Something goes wrong while init"),
TEXT("Oops!"), MB_OK);
        }
        break;


    case WM_COMMAND:
        if (lParam == (LPARAM)hBtnStartRecording) {
            if (BtnStartRecordingClick() == EXIT_FAILURE) {
                MessageBox(hWnd, TEXT("Something goes wrong while
recording"), TEXT("Oops!"), MB_OK);
            }
        }
        else if (lParam == (LPARAM)hBtnStopRecording) {
            if (BtnStopRecordingClick() == EXIT_FAILURE) {
                MessageBox(hWnd, TEXT("Something goes wrong while
stopping."), TEXT("Oops!"), MB_OK);
            }
        }
        else if (lParam == (LPARAM)hBtnReplay) {
            int err = 0;
            if ((err = BtnReplayClick()) != EXIT_SUCCESS) {
                MessageBox(hWnd, TEXT("Something goes wrong while
replaying."), TEXT("Oops!"), MB_OK);
            }
        }
        else if (lParam == (LPARAM)hBtnRecognize) {
            if (BtnRecognizeClick() == EXIT_FAILURE) {
            MessageBox(hWnd, TEXT("Something goes wrong while recognizing."),
TEXT("Oops!"), MB_OK);
            }
        }
        break;

    case WM_DESTROY:
        err = BassDllCleanup();
        if (err != 0) {
            MessageBox(hWnd, TEXT("Something goes wrong while cleanup"),
TEXT("Oops!"), MB_OK);
        }
        PostQuitMessage(EXIT_SUCCESS);
        break;

    default:
        return DefWindowProcA(hWnd, uMsg, wParam, lParam);
    }
    return 0;
};
```

Vlazam.cpp

```
#ifndef _CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS
#endif

#include <Windows.h>
#include <fstream>
#include <vector>
#include <string>
#include "Vlazam.h"
#include "../Dependencies/kiss_fft130/tools/kfc.h"
#include "../Dependencies/bass/bass.h"

// ranges of frequencies
const int RANGE_COUNT = 5;
const int RANGE[RANGE_COUNT] = { 40, 80, 120, 180, 300 };
// stuff for recording
char* recBuf = nullptr;  // buffer for recording
DWORD recLen;         // size of this buffer
HRECORD rchan = 0;
HSTREAM chan = 0;
BOOL CALLBACK recordingCallback(HRECORD handle, const void* buffer, DWORD
length, void* user);

int loadSongs(SongHash* &songsHashes, const char* dirPath, int &numOfSongs) {
        WIN32_FIND_DATA data;
        HANDLE hFind;

        char* correctDirPath = nullptr;
        if (dirPath[strlen(dirPath) - 1] != '*') {
                correctDirPath = (char*)malloc(sizeof(char) * (strlen(dirPath)
+ 2));
                strcpy_s(correctDirPath, sizeof(char) * (strlen(dirPath) + 2),
dirPath);
                strcat_s(correctDirPath, sizeof(char) * (strlen(dirPath) + 2),
"*");
        }
        else {
                correctDirPath = (char*)malloc(sizeof(char) * (strlen(dirPath)
+ 1));
                strcpy_s(correctDirPath, sizeof(char) * (strlen(dirPath) + 1),
dirPath);
        }
        std::vector<char*> vect;
        if ((hFind = FindFirstFile(correctDirPath, &data)) ==
INVALID_HANDLE_VALUE) {
                return -1;
        }

        do {
                if (data.cFileName[0] != '.') {
                        int buflen = strlen(data.cFileName) +
strlen(DB_DIRECTORY_PATH) + 1;
                        char* buf = (char*)malloc(sizeof(char) * buflen);
                        strcpy_s(buf, sizeof(char) * buflen,
DB_DIRECTORY_PATH);
                        strcat_s(buf, sizeof(char) * buflen, data.cFileName);
                        vect.push_back(buf);
                }
        } while (FindNextFile(hFind, &data) != 0);
        FindClose(hFind);
```

```
	numOfSongs = vect.size();
	songsHashes = (SongHash*)malloc(sizeof(SongHash) * numOfSongs);
	if (!songsHashes) {
		return -1;
	}
	//songsHashes = new SongHash[numOfSongs];
	memset(songsHashes, 0, sizeof(SongHash) * numOfSongs);

	for (int i = 0; i < numOfSongs; i++) {
		int k = sizeof(long);
		songsHashes[i].size = fileSize(vect[i]) / k;
		songsHashes[i].buffer = (long*)malloc(sizeof(long) *
songsHashes[i].size);
		songsHashes[i].songName = (char*)malloc(strlen(vect[i]) + 1);
		if (!((songsHashes[i].buffer) || (!songsHashes[i].songName)))
{
			return -1;
		}
		strcpy_s(songsHashes[i].songName, strlen(vect[i]) + 1,
vect[i]);


		if (!songsHashes[i].buffer) {
			return -1;
		}
		memset(songsHashes[i].buffer, 0, songsHashes[i].size *
sizeof(long));
		//std::ifstream file(vect[i].c_str());
		//if (!file.is_open()) {
		//	continue;
		//	// return -1??
		//}
		//file.read(songsHashes[i].buffer, )
		FILE* fp;
		if (fopen_s(&fp, vect[i], "rb")) {
			return -1;
		}
		int res = fread(songsHashes[i].buffer, sizeof(long),
songsHashes[i].size, fp);
		// fread(songsHashes[i].buffer, songsHashes[i].size, 1, fp);
		fclose(fp);
	}
	return 0;
}
// returns 0 if success
int changeFileExtension(char *&fileName, const char* newExtension) {
	int len = strlen(fileName);
	int i = len;
	while ((i >= 0) && (fileName[i] != '.')) {
		i--;
	}
	if (i < 0) {
		return -1;
	}
	int newLen = i + strlen(newExtension) + 1;
	char* buf = (char*)malloc(newLen);
	if (!buf) {
		return -1;
	}
	strcpy_s(buf, newLen, fileName);
	strcpy_s(buf + i, newLen, newExtension);
```

```
        fileName = buf;

        return 0;
}

// returns -1 if error; otherwise returns 0
int saveToFile(const char* fileName, const char* buf, int bufLen) {
        std::ofstream file(fileName);
        if (!file) {
                return -1;
        }
        file.write(buf, bufLen);

        return 0;
}

long hash(long p1, long p2, long p3, long p4) {
        return  (p1 - (p1 % FUZZ_FACTOR)) +
                (p2 - (p2 % FUZZ_FACTOR)) * 100 +
                (p3 - (p3 % FUZZ_FACTOR)) * 10000 +
                (p4 - (p4 % FUZZ_FACTOR)) * 10000000;
}

size_t fileSize(const char* fileName) {
        //
                at the end
        std::ifstream file(fileName, std::ifstream::ate |
std::ifstream::binary);
        int res = static_cast<int>(file.tellg());
        file.close();
        return res;
}

int getIndex(const int freq) {
        int i = 0;
        const int* range = RANGE;

        while (range[i] < freq) {
                i++;
        }
        return i;
}

char* intToCharptr(long long a) {
        // max long long 9223372036854775807
        const int maxSize = 20;

        char* result = (char*)malloc(maxSize);
        memset(result, 0, maxSize);

        a = abs(a);
        int buf = a % 10;

        do {
                buf = a % 10;
                a /= 10;
                switch (buf) {
                case 0:
                        strcat_s(result, maxSize, "0");
                        break;
                case 1:
                        strcat_s(result, maxSize, "1");
```

```
                        break;
                case 2:
                        strcat_s(result, maxSize, "2");
                        break;
                case 3:
                        strcat_s(result, maxSize, "3");
                        break;
                case 4:
                        strcat_s(result, maxSize, "4");
                        break;
                case 5:
                        strcat_s(result, maxSize, "5");
                        break;
                case 6:
                        strcat_s(result, maxSize, "6");
                        break;
                case 7:
                        strcat_s(result, maxSize, "7");
                        break;
                case 8:
                        strcat_s(result, maxSize, "8");
                        break;
                case 9:
                        strcat_s(result, maxSize, "9");
                        break;
                }
        }
        while (a != 0);

        int j = strlen(result) - 1;
        char bufc;
        for (int i = 0; i < (j / 2) + 1; i++) {
                bufc = result[i];
                result[i] = result[j - i];
                result[j - i] = bufc;
        }

        return result;
}

char* getFileName(const char* fullPath) {
        int len = strlen(fullPath);
        int i = len - 1;
        while ((i >= 0) && (fullPath[i] != '\\') && (fullPath[i] != '/')) {
                i--;
        }
        char* result = (char*)malloc(len - i);
        if (!result) {
                return nullptr;
        }
        strcpy_s(result, len - i, fullPath + i + 1);
        return result;
}

char* getFileNameWithoutExtension(const char* fullPath) {
        int len = strlen(fullPath);
        int i = len - 1;
        int j = len - 1;
        while ((i >= 0) && (fullPath[i] != '\\') && (fullPath[i] != '/')) {
                i--;
        }
        while ((j >= 0) && (j >= i) && (fullPath[j] != '.')) {
```

```
                j--;
        }
        char* buf = (char*)malloc(strlen(fullPath) * sizeof(char));
        if (!buf) {
                return nullptr;
        }
        strcpy_s(buf, len + 1, fullPath);
        buf[j] = '\0';

        char* result = (char*)malloc(j - i + 1);
        if (!result) {
                return nullptr;
        }
        strcpy_s(result, len - i + 1, buf + i + 1);
        return result;
}

int addToDatabase(const char* fileName) {

        int sizeWithoutHeaders = fileSize(fileName) - 44;
        char* audio = (char*)malloc(sizeof(char) * sizeWithoutHeaders);
        if (!audio) {
                // out of memory
                return -1;
        }

        // read all audio meat
        FILE* fp;
        if (fopen_s(&fp, fileName, "rb")) {
                // Cannot open a file
                return -1;
        }
        fseek(fp, 44, 0);
        int length = fread(audio, 1, sizeWithoutHeaders, fp);
        fclose(fp);

        const int chunkSize = CHUNK_SIZE;
        int sampledChunkCount = length / chunkSize;

        // allocate memory for result complex array
        kiss_fft_cpx** result = (kiss_fft_cpx**)malloc(sizeof(kiss_fft_cpx*) *
sampledChunkCount);
        for (int j = 0; j < sampledChunkCount; j++) {
                result[j] = (kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) *
chunkSize);
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < chunkSize; i++) {
                        result[j][i].r = 0;
                        result[j][i].i = 0;
                }
        }

        for (int j = 0; j < sampledChunkCount; j++) {
                kiss_fft_cpx* complexArray =
(kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) * chunkSize);
                if (!complexArray) {
                        return -1;
                }

                for (int i = 0; i < chunkSize; i++) {
                        kiss_fft_cpx bufCpx;
```

```
                        bufCpx.r = audio[j * chunkSize + i];
                        bufCpx.i = 0;
                        complexArray[i] = bufCpx;
                }

                kfc_fft(chunkSize, complexArray, result[j]);
        }

        // create digital signature
        // array for recording high values of amplittude
        double** highscores = (double**)malloc(sizeof(double*) *
sampledChunkCount);
        if (!highscores) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                highscores[j] = (double*)malloc(sizeof(double) * 5);
                if (!highscores[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        highscores[j][i] = 0;
                }
        }
        // array for recording frequences of this values
        int** points = (int**)malloc(sizeof(int*) * sampledChunkCount);
        if (!points) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                points[j] = (int*)malloc(sizeof(int) * RANGE_COUNT);
                if (!points[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        points[j][i] = 0;
                }
        }

        SongHash songHash;
        songHash.size = sampledChunkCount;
        songHash.buffer = (long*)malloc(sizeof(long) * songHash.size);
        songHash.songName = (char*)malloc(strlen(RECORDED_BUF_FILENAME) + 1);
        if (!((songHash.buffer) || (!songHash.songName))) {
                return -1;
        }
        strcpy_s(songHash.songName, strlen(RECORDED_BUF_FILENAME) + 1,
RECORDED_BUF_FILENAME);
        // values of recording sample
        for (int t = 0; t < sampledChunkCount; t++) {
                for (int freq = 40; freq < 300; freq++) {
                        double mag = log(abs(result[t][freq].r) + 1);
                        int index = getIndex(freq);
                        if (mag > highscores[t][index]) {
                                highscores[t][index] = mag;
                                points[t][index] = freq;
                        }
                }
```

```
                songHash.buffer[t] = hash(points[t][0], points[t][1],
points[t][2], points[t][3]);
        }

        char* buf = getFileName(fileName);
        int outputFileNameLen = strlen(DB_DIRECTORY_PATH) + strlen(buf) + 1;
        char* outputFileName = (char*)malloc(outputFileNameLen);
        if (!outputFileName) {
                return -1;
        }
        strcpy_s(outputFileName, outputFileNameLen, DB_DIRECTORY_PATH);
        strcat_s(outputFileName, outputFileNameLen, buf);
        changeFileExtension(outputFileName, ".bin");

        FILE* fo;
        if (fopen_s(&fo, outputFileName, "wb")) {
                return -1;
        }
        fwrite(songHash.buffer, sampledChunkCount, 1, fo);
        fclose(fo);
        return 0;
}

// returns -1 if error
int startRecording() {
        WAVEFORMATEX* wf;
        if (recBuf) {
                // free old recording
                BASS_StreamFree(chan);
                chan = 0;
                free(recBuf);
                recBuf = nullptr;
                // close output device before recording in case of half-duplex
device
                // BASS_Free();
        }
        // allocate initial buffer and make space for WAVE header
        recBuf = (char*)malloc(BUFSTEP);
        memset(recBuf, 0, BUFSTEP);
        recLen = 44;

        // fill the WAVE header
        memcpy(recBuf, "RIFF\0\0\0\0WAVEfmt \20\0\0\0", 20);
        memcpy(recBuf + 36, "data\0\0\0\0", 8);
        wf = (WAVEFORMATEX*)(recBuf + 20);
        wf->wFormatTag = 1;
        wf->nChannels = CHANS;
        wf->wBitsPerSample = 8;
        wf->nSamplesPerSec = FREQ;
        wf->nBlockAlign = wf->nChannels * wf->wBitsPerSample / 8;
        wf->nAvgBytesPerSec = wf->nSamplesPerSec * wf->nBlockAlign;
        // start recording
        rchan = BASS_RecordStart(FREQ, CHANS, BASS_SAMPLE_8BITS,
recordingCallback, 0);
        if (!rchan) {
                free(recBuf);
                recBuf = NULL;
                return -1;
        }
        return 0;
}
```

```
// buffer the recorded data
BOOL CALLBACK recordingCallback(HRECORD handle, const void* buffer, DWORD
length, void* user)
{
	// increase buffer size if needed
	if ((recLen % BUFSTEP) + length >= BUFSTEP) {
		recBuf = (char*)realloc(recBuf, ((recLen + length) / BUFSTEP +
1) * BUFSTEP);
		if (!recBuf) {
			rchan = 0;
			return FALSE; // stop recording
		}
	}
	// buffer the data
	memcpy(recBuf + recLen, buffer, length);
	recLen += length;
	return TRUE; // continue recording
}

// returns 0 if ok
int stopRecording() {
	int result = 0;
	if (!BASS_ChannelStop(rchan)) {
		result = -1;
	}
	rchan = 0;

	*(DWORD*)(recBuf + 4) = recLen - 8;
	*(DWORD*)(recBuf + 40) = recLen - 44;

	int err = 0;

	return result;
}

int playFileWAV(const char* fileName) {
	chan = BASS_StreamCreateFile(FALSE, fileName, 0, 0, 0);

	BASS_ChannelPlay(chan, TRUE);
	return 0;
}

void waitTillPlaying() {
	while ((BASS_ChannelIsActive(chan) == BASS_ACTIVE_PLAYING)) {
		Sleep(100);
	}
}

int BassDllInit() {
	if (!BASS_Init(DEFAULT_DEVICE, FREQ, BASS_DEVICE_3D, 0, NULL)) {
		return BASS_ErrorGetCode();
	}
	if (!BASS_RecordInit(DEFAULT_DEVICE)) {
		return BASS_ErrorGetCode();
	}
	return 0;
}

int BassDllCleanup() {
	if (BASS_IsStarted()) {
		if (!BASS_Free()) {
			return BASS_ErrorGetCode();
```

```
                }
                if (!BASS_RecordFree()) {
                        return BASS_ErrorGetCode();
                }
        }
        return 0;
}

int saveRecording(const char* fileName) {
        int res = saveToFile(fileName, recBuf, recLen);
        free(recBuf);
        recBuf = nullptr;
        return res;
}

int recognizeSample(char**& resultSongs, int& countSongs) {
        //////////////////
        // all from addToDB
        int sizeWithoutHeaders = fileSize(RECORDED_BUF_FILENAME) - 44;
        char* audio = (char*)malloc(sizeof(char) * sizeWithoutHeaders);
        if (!audio) {
                // out of memory
                return -1;
        }

        // read all audio meat
        FILE* fp;
        if (fopen_s(&fp, RECORDED_BUF_FILENAME, "rb")) {
                // Cannot open a file
                return -1;
        }
        fseek(fp, 44, 0);
        int length = fread(audio, 1, sizeWithoutHeaders, fp);
        fclose(fp);

        const int chunkSize = CHUNK_SIZE;
        int sampledChunkCount = length / chunkSize;

        // allocate memory for result complex array
        kiss_fft_cpx** result = (kiss_fft_cpx**)malloc(sizeof(kiss_fft_cpx*) *
sampledChunkCount);
        for (int j = 0; j < sampledChunkCount; j++) {
                //result[j] = (kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) *
chunkSize);
                result[j] = new kiss_fft_cpx[chunkSize];
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < chunkSize; i++) {
                        result[j][i].r = 0;
                        result[j][i].i = 0;
                }
        }

        for (int j = 0; j < sampledChunkCount; j++) {
                kiss_fft_cpx* complexArray =
(kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) * chunkSize);
                if (!complexArray) {
                        return -1;
                }

                for (int i = 0; i < chunkSize; i++) {
                        kiss_fft_cpx bufCpx;
```

```
                        bufCpx.r = audio[j * chunkSize + i];
                        bufCpx.i = 0;
                        complexArray[i] = bufCpx;
                }

                kfc_fft(chunkSize, complexArray, result[j]);
        }
        //free(audio);
        // create digital signature
        // array for recording high values of amplittude
        double** highscores = (double**)malloc(sizeof(double*) *
sampledChunkCount);
        if (!highscores) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                highscores[j] = (double*)malloc(sizeof(double) * 5);
                if (!highscores[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        highscores[j][i] = 0;
                }
        }
        // array for recording frequences of this values
        int** points = (int**)malloc(sizeof(int*) * sampledChunkCount);
        if (!points) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                points[j] = (int*)malloc(sizeof(int) * RANGE_COUNT);
                if (!points[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        points[j][i] = 0;
                }
        }

        SongHash songHash;
        songHash.size = sampledChunkCount;
        songHash.buffer = (long*)malloc(sizeof(long) * songHash.size);
        songHash.songName = (char*)malloc(strlen(RECORDED_BUF_FILENAME) + 1);
        if (!((songHash.buffer) || (!songHash.songName))) {
                return -1;
        }
        strcpy_s(songHash.songName, strlen(RECORDED_BUF_FILENAME) + 1,
RECORDED_BUF_FILENAME);
        // values of recording sample
        for (int t = 0; t < sampledChunkCount; t++) {
                for (int freq = 40; freq < 300; freq++) {
                        double mag = log(abs(result[t][freq].r) + 1);
                        int index = getIndex(freq);
                        if (mag > highscores[t][index]) {
                                highscores[t][index] = mag;
                                points[t][index] = freq;
                        }
                }
```

```
                songHash.buffer[t] = hash(points[t][0], points[t][1],
points[t][2], points[t][3]);
        }
        //
        // time for free
        for (int j = 0; j < sampledChunkCount; j++) {
                free(result[j]);
        }

        free(result);

        for (int j = 0; j < sampledChunkCount; j++) {
                free(highscores[j]);
        }
        free(highscores);
        // end addToDB
        // todo
        // 1. getSongsFromDBCount
        // 2. loadSong
        SongHash* songs = nullptr;
        int songsCount = 0;
        int res = loadSongs(songs, DB_DIRECTORY_PATH, songsCount);
        if (res == -1) {
                return -1;
        }
        int* collisions = (int*)malloc(sizeof(int) * songsCount);
        if (!collisions) {
                return -1;
        }
        memset(collisions, 0, sizeof(int) * songsCount);
        for (int i = 0; i < songsCount; i++) {
                for (int z = 0; z < sampledChunkCount; z++) {
                        for (int w = 0; w < sampledChunkCount; w++) {
                                if (songs[i].buffer[z] == songHash.buffer[w]) {
                                        collisions[i]++;
                                }
                        }
                }
        }

        int maxIndex = 0;
        int maxValue = collisions[maxIndex];
        for (int i = 1; i < songsCount; i++) {
                if (collisions[i] > collisions[maxIndex]) {
                        maxIndex = i;
                        maxValue = collisions[maxIndex];
                }
        }
        // todo: return more than 1 song, if delta = 50%
        int suitableCount = 1;
        char** suitableSongNames = (char**)malloc(suitableCount);
        if (!suitableSongNames) {
                return -1;
        }
                                                                // sizeof(char*)
* ??
        suitableSongNames[0] =
getFileNameWithoutExtension(songs[maxIndex].songName);
        resultSongs = suitableSongNames;
        countSongs = suitableCount;

        free(songs);
```

```
	return 0;
}

int getSampleHash(char* sampleFileName, SongHash &songHash) {
	int sizeWithoutHeaders = fileSize(sampleFileName) - 44;
	//char* audio = (char*)malloc(sizeof(char) * sizeWithoutHeaders);
	char* audio = new char[sizeWithoutHeaders];
	if (!audio) {
		// out of memory
		return -1;
	}

	// read all audio meat
	FILE* fp;
	if (fopen_s(&fp, sampleFileName, "rb")) {
		// Cannot open a file
		return -1;
	}
	// miss headers
	fseek(fp, 44, 0);
	int length = fread(audio, 1, sizeWithoutHeaders, fp);
	fclose(fp);

	const int chunkSize = CHUNK_SIZE;
	int sampledChunkCount = length / chunkSize;

	// allocate memory for result(OUTPUT) complex array
	//kiss_fft_cpx** result = (kiss_fft_cpx**)malloc(sizeof(kiss_fft_cpx*)
* sampledChunkCount);
	kiss_fft_cpx** result = new kiss_fft_cpx * [sampledChunkCount];
	for (int j = 0; j < sampledChunkCount; j++) {
		//result[j] = (kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) *
chunkSize);
		result[j] = new kiss_fft_cpx[chunkSize];
	}
	for (int j = 0; j < sampledChunkCount; j++) {
		for (int i = 0; i < chunkSize; i++) {
			result[j][i].r = 0;
			result[j][i].i = 0;
		}
	}

	// allocate memory for result(INPUT) complex array
	// kiss_fft_cpx* complexArray =
(kiss_fft_cpx*)malloc(sizeof(kiss_fft_cpx) * chunkSize);
	kiss_fft_cpx* complexArray = new kiss_fft_cpx[chunkSize];
	if (!complexArray) {
		return -1;
	}
	// and initialize it
	for (int i = 0; i < chunkSize; i++) {
		kiss_fft_cpx bufCpx;
		bufCpx.r = audio[j * chunkSize + i];
		bufCpx.i = 0;
		complexArray[i] = bufCpx;
	}

	// performing fast fourie transform
	for (int j = 0; j < sampledChunkCount; j++) {
		kfc_fft(chunkSize, complexArray, result[j]);
	}
```

```
        delete[] complexArray;
        delete[] audio;
        // ends with Fourie Transform

        // now create sing signature
        // array for recording high values of amplittude
        //double** highscores = (double**)malloc(sizeof(double*) *
sampledChunkCount);
        double** highscores = new double* [sampledChunkCount];
        if (!highscores) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                //highscores[j] = (double*)malloc(sizeof(double) * 5);
                highscores[j] = new double[RANGE_COUNT];
                if (!highscores[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        highscores[j][i] = 0;
                }
        }
        // array for recording frequences of this values
        //int** points = (int**)malloc(sizeof(int*) * sampledChunkCount);
        int** points = new int* [sampledChunkCount];
        if (!points) {
                return -1;
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                //points[j] = (int*)malloc(sizeof(int) * RANGE_COUNT);
                points[j] = new int[RANGE_COUNT];
                if (!points[j]) {
                        return -1;
                }
        }
        for (int j = 0; j < sampledChunkCount; j++) {
                for (int i = 0; i < RANGE_COUNT; i++) {
                        points[j][i] = 0;
                }
        }
        // now create SongHash object and initialize it
        songHash.size = sampledChunkCount;
        //songHash.buffer = (long*)malloc(sizeof(long) * songHash.size);
        songHash.buffer = new long[songHash.size];
        //songHash.songName = (char*)malloc(strlen(RECORDED_BUF_FILENAME) +
1);
        songHash.songName = new char[strlen(RECORDED_BUF_FILENAME) + 1];
        if (!((songHash.buffer) || (!songHash.songName))) {
                return -1;
        }
        strcpy_s(songHash.songName, strlen(RECORDED_BUF_FILENAME) + 1,
RECORDED_BUF_FILENAME);
        // values of recording sample
        for (int t = 0; t < sampledChunkCount; t++) {
                for (int freq = 40; freq < 300; freq++) {
                        double mag = log(abs(result[t][freq].r) + 1);
                        int index = getIndex(freq);
                        if (mag > highscores[t][index]) {
                                highscores[t][index] = mag;
                                points[t][index] = freq;
```

```
                        }
                }
                songHash.buffer[t] = hash(points[t][0], points[t][1],
points[t][2], points[t][3]);
        }

        // free memory
        for (int j = 0; j < sampledChunkCount; j++) {
                delete[] result[j];
        }
        delete[] result;

        for (int j = 0; j < sampledChunkCount; j++) {
                delete[] highscores[j];
        }
        delete[] highscores;

        return 0;
}
```

SongHash.cpp

```
#include "SongHash.h"

SongHash::SongHash() {
        this->buffer = nullptr;
        this->songName = nullptr;
        this->size = 0;
}

SongHash::SongHash(long* bufptr, char* songname, size_t bufsize) {
        this->buffer = bufptr;
        this->songName = songname;
        this->size = bufsize;
}
```

Обозначение	Наименование	Дополнительные сведения
	Текстовые документы	
БГУИР КР 1–40 01 01 619 ПЗ	Пояснительная записка	45 с.
	Графические документы	
ГУИР 8510006 623 СП	"Программное средство для распознавания музыки", А1, схема программы, чертеж	Формат А1

					БГУИР КР 1-40 01 01 623 Д1			
Изм.	Л.	№ докум.	Подп.	Дата			Лист	Листов
Разраб.		Петровец В.Н.			Программное средство для распознавания музыки	Т	45	45
Пров.		Жиденко А.Л.						
					Ведомость курсовой работы	Кафедра ПОИТ гр. 851006		