# Содержание

[Содержание 5](#_Toc480836594)

[ВВЕДЕНИЕ 7](#_Toc480836595)

[**1** ОБЗОР ЛИТЕРАТУРЫ 9](#_Toc480836596)

[**2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 17](#_Toc480836597)

[**2.1** Выбор аппаратной платформы 17](#_Toc480836598)

[**2.2** Выбор целевой платформы и средств разработки 18](#_Toc480836599)

[**2.3** Планирование логических модулей 20](#_Toc480836600)

[**2.3.1** Модуль взаимодействия с базой данных 21](#_Toc480836601)

[**2.3.2** Модуль взаимодействия с файловой системой 21](#_Toc480836602)

[**2.3.3** Модуль алгоритмов дискретизации 22](#_Toc480836603)

[**2.3.4** Модуль реализации бизнес-логики 22](#_Toc480836604)

[**2.3.5** Модуль прослушивания подключенных устройств 22](#_Toc480836605)

[**2.3.6** Модуль генерации команд, для управления драйвером 22](#_Toc480836606)

[**2.3.7** Модуль взаимодействия с драйвером 22](#_Toc480836607)

[**2.3.8** Модуль взаимодействия с драйвером 22](#_Toc480836608)

[**3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 23](#_Toc480836609)

[**3.1** Описание принципов работы .Net фреймворк 23](#_Toc480836610)

[**3.2** Описание работы модуля взаимодействия с базой данных 26](#_Toc480836611)

[**3.3** Работа модуля дискретизации 32](#_Toc480836618)

[**3.4** Работа модуля реализации бизнес-логики 34](#_Toc480836619)

[**3.5** Работа модуля прослушивания подключенных устройств 35](#_Toc480836620)

[**3.7** Модуль взаимодействия с драйвером 37](#_Toc480836621)

[**3.8** Модуль построения пользовательского интерфейса 38](#_Toc480836622)

[**3.9** Модуль взаимодействия с файловой системой 40](#_Toc480836623)

[**3.10** Принцип инверсии управления 41](#_Toc480836624)

[**3.11** Разработка функциональной схемы 42](#_Toc480836625)

[**4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 47](#_Toc480836626)

[**4.1** Алгоритм преобразования входных музыкальных файла 47](#_Toc480836627)

[**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ 50](#_Toc480836628)

[**5.1** Выбор средств и методологии тестирования 50](#_Toc480836629)

[**5.2** Модульное тестирование 52](#_Toc480836630)

[**5.3** Интеграционное тестирование 53](#_Toc480836632)

[**5.4** Системное тестирование 53](#_Toc480836633)

[ЗАКЛЮЧЕНИЕ 55](#_Toc480836635)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 56](#_Toc480836636)

[ПРИЛОЖЕНИЕ А 57](#_Toc480836638)

[ПРИЛОЖЕНИЕ Б 58](#_Toc480836639)

# ВВЕДЕНИЕ

Современные технологии развиваются с очень большой скоростью. Закон Мура гласит, что количество транзисторов, расположенных на интегральной схеме удваивается каждые два года. Также осуществляется рост размеров операционных систем и программ. К примеру, размер операционной системы, Windows 95 составлял примерно 50 мегабайт, размер операционной системы Windows 8 составляет около 15 гигабайт, что примерно в 300 раз больше предшественника. Из этого следует, что объемы запоминающих устройств, а также скорость чтения и записи информации тоже растут.

В конце 20 века для хранения информации начали активно использоваться гибкие магнитные диски. Они отличались большим на тот момент объемом памяти, большой скоростью взаимодействия и высокой степенью практичности. К тому же у дискеты была особенность, которую не имели на тот момент другие носители, она позволяла механически включать и выключать режим защиты от записи. В 2011 году компания Sony прекратила продажу и выпуск гибких магнитных дисков, а в 2014 компания Toshiba организовала на заводе по выпуску дискет овощную ферму. И только в 2016 году правительственная организация Пентагон решил полностью прекратить использование гибких магнитных дисков к 2018 году.

За 35 лет активного использования данной технологии было создано огромное количество гибких магнитных дискет. Дискета является самым продаваемым носителем информации, благодаря ее низкой стоимости, и не удивителен тот факт, что для поддержки старой технологии до сих пор выпускаются накопители на гибких магнитных дисках, правда в современном варианте они подключаются по интерфейсу USB. И если учесть, что в современном мире для хранения информации уже давно используются USB-носители, переносные жесткие диски и облака, можно прийти к выводу, что на данный момент имеется огромное количество неиспользуемых гибких магнитных дисков и накопителей для них, в этом и заключается актуальность дипломного проекта.

Целью данного дипломного проекта является проектирование и реализация программно-аппаратного комплекса, который при помощи устаревших технологий ввода-вывода, таких как накопители гибких дисков, смог бы воспроизводить звуковые композиции.

Для достижения данной цели необходимо выполнить следующие задачи:

* реализовать плату, при помощи которой можно будет легко увеличивать количество подключенных устройств ввода-вывода;
* подключить к спроектированной плате микроконтроллер, который будет выполнять функцию драйвера;
* реализовать прошивку для драйвера;
* реализовать интерфейс и методы обмена данными между драйвером и персональным компьютером;
* реализовать алгоритм дискретизации и преобразование в формат MIDI звуковых файлов;
* реализовать алгоритм преобразования MIDI- файлов в набор команд для драйвера;
* реализовать пользовательский интерфейс приложения, который будет прост в использовании;
* провести тестирование реализованного программно-аппаратного комплекса.

# **1** ОБЗОР ЛИТЕРАТУРЫ

Перед началом разработки дипломного проекта, нужно определиться с основными понятиями в теории звука.

Звуковая волна является [физическим явление](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%B7%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D1%8F%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5)м, представляющим собой распространение в виде [упругих волн](https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D1%83%D0%B3%D0%B8%D0%B5_%D0%B2%D0%BE%D0%BB%D0%BD%D1%8B) механических колебаний. Звуковые волны возникают благодаря вибрированию тел, и распространяются в упругих средах, таких как: твёрдое тело, жидкость, газ. В узком смысле под звуком понимают колебания и характеристики того, как они воспринимаются органами человека и животных. В связи с тем, что звуковая волна имеет колебательную природу, она характеризуется такими величинами как амплитуда, частота и фаза [1].

Частота измеряется в герцах и обозначает количество колебаний за период времени. Чем больше звуковых колебаний совершается за секунду, тем более высокое звучание будет восприниматься живым организмом.

Следующей рассматриваемой характеристикой звуковой волны, является длина волны. Длина волны – это расстояние между двумя ее ближайшими точками, движущимися в одной фазе. Поскольку колеблющиеся точки оказываются в одинаковом положении через время, равное периоду колебаний, то длина волны, это расстояние, на которое она распространяется за время, равное одному периоду.

Амплитудой волны является величина, показывающая наибольшее звуковое давление при сгущениях и разряжениях, она характеризует интенсивность звука.

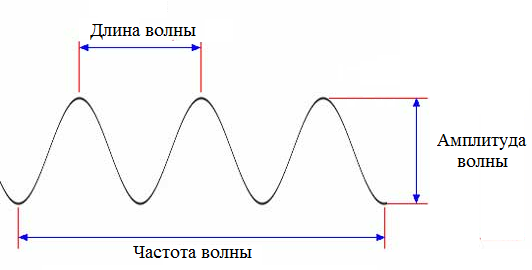


Рисунок 1.1 – Звуковая волна

На рисунке 1.1 проиллюстрированы длина и частота волны.

В музыкальной терминологии звука существуют такие важные обозначения, как октава, тон и обертон звука. Октава – [музыкальный интервал](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%B2%D0%B0%D0%BB_(%D0%BC%D1%83%D0%B7%D1%8B%D0%BA%D0%B0)), в котором соотношение [частот](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%BE%D1%82%D0%B0) между звуками составляет один к двум (то есть частота высокого звука в 2 раза больше низкого). На слух октава воспринимается как стабильный, базисный музыкальный интервал. Октава в свою очередь состоит из тонов и обертонов.

В музыке используется крайне малое число тонов. Объясняется это из соображений принципа гармонической созвучности, всё основано на принципе октав.

Рассмотрим теорию музыкальных тонов на примере струны, натянутой определённым образом. Такая струна, в зависимости от силы натяжения, будет иметь «настройку» на какую-то одну конкретную частоту. При воздействии на эту струну с одной определённой силой, вызовет её колебания и будет наблюдаться один определенный тон звука, мы услышим основной тон.

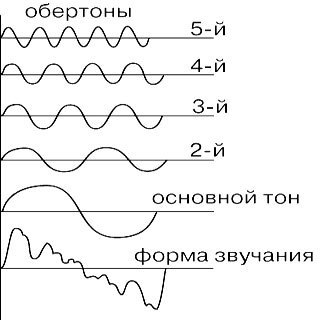


Рисунок 1.2 – Колебания идеальной струны

На рисунке 1.2 изображены составляющие колебания идеальной струны: 1 – основной тон, 2-5 – гармоники, соответствующие первому — четвёртому обертонам.

Однако, большинство музыкальных инструментов никогда не воспроизводят одни чистые основные тона, их неизбежно сопровождают обертонами(призвуки).

Обертоны бывают гармоническими и негармоническими. Частоты гармонических обертонов кратны частоте основного тона (гармонические обертоны вместе с основным тоном также называются гармониками). В музыке практически исключается использование некратных обертонов, поэтому термин сводится к понятию обертон, подразумевая под собой гармонический.

Тембр – это свойство музыкальных звуков, который придает музыкальным инструментам и голосам их уникальную специфику звучания, даже если сравнивать звуки одинаковой высоты и громкости. Тембр каждого музыкального инструмента зависит от распределения звуковой энергии по обертонам в момент появления звука [2].

Поскольку на практике практически не встречаются волны одной частоты, то возникает необходимость разложения всего звукового спектра слышимого диапазона на обертоны или гармоники, такой метод называется Фурье-анализом. Для этих целей существуют графики, которые отображают зависимость относительной энергии звуковых колебаний от частоты. Такой график называется графиком частотного спектра звука (см. рисунок 1.3). Частотный спектр звука бывает двух типов: дискретный и непрерывный. Дискретный график спектра отображает частоты по отдельности, разделённые пустыми промежутками. В непрерывном спектре присутствуют сразу все звуковые частоты.

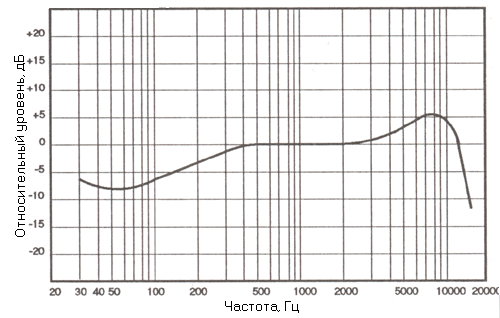


Рисунок 1.3 – Пример графика частотного спектра звука

В случае с музыкой или акустикой чаще всего используется обычный график Амплитудно-Частотой характеристики (см. рисунок 1.4). Глядя на такой график легко понять, например, сильные или слабые стороны конкретного динамика или акустической системы в целом, наиболее сильные участки энергетической отдачи, частотные спады и подъёмы, затухания, а также проследить крутизну спада.

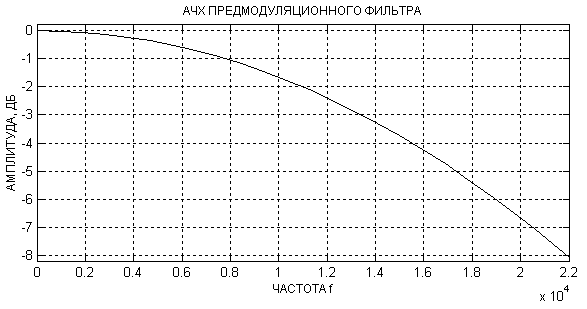


Рисунок 1.4 – Пример графика АЧХ

Теперь, когда мы разобрались с основными понятиями теории звука, необходимо понять, как устройства ввода – вывода могут воспроизводить мелодии. Рассмотрим данный вопрос на примере накопителей гибких дисков.

Накопитель гибких дисков – уже устаревшее устройство: сегодня практически никто не использует дискеты как носитель информации, поскольку существуют куда более удобные и быстрые способы распространения информации. Работающие образцы накопителей гибких дисков могут быть использованы не по первоначальному назначению, например, для извлечения из них звуковых волн. Накопители гибких дисков достаточно просты: они получают команду, когда нужно вращать диск, когда нужно читать или писать и как далеко сдвинуть магнитную головку. Мелодии извлекаются из накопителей гибких дисков через управление скоростью читающей головки, как правило, используется MIDI-запись нужной мелодии [3]. На рисунке 1.5 изображена схема накопителя гибких дисков. Используя какую-либо плату управления и записанный музыкальный алгоритм возможно получить столь голосную полифонию, сколько используется дисководов. Для управления устройством используется любой микроконтроллер, который может посылать управляющие сигналы на накопитель в нужной последовательности, например, Arduino или Raspberry. Управление может осуществляться, как и «hardware», когда код мелодии загружается непосредственно в контроллер, так и «software» – когда мелодия обрабатывается специальной программой на компьютере, а контроллер выступает в роли драйвера, распределяя и передавая полученный сигнал на дисководы.

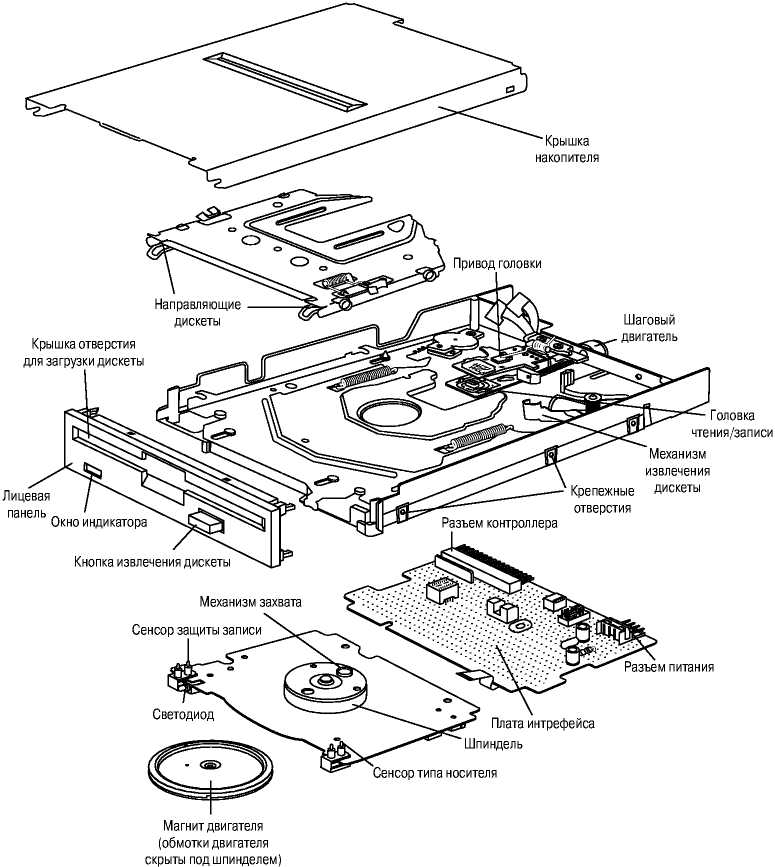


Рисунок 1.5 – Схема накопителя гибких дисков

MIDI – [стандарт](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82) [цифровой звукозаписи](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%BE%D0%B9_%D0%B7%D0%B2%D1%83%D0%BA) на формат обмена данными между [электронными музыкальными инструментами](https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D0%BC%D1%83%D0%B7%D1%8B%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B8%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D1%8B). Интерфейс позволяет единообразно кодировать в цифровой форме такие данные как нажатие клавиш, настройку [громкости](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%BE%D0%BC%D0%BA%D0%BE%D1%81%D1%82%D1%8C_%D0%B7%D0%B2%D1%83%D0%BA%D0%B0) и других акустических параметров, выбор [тембра](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BC%D0%B1%D1%80), [темпа](https://ru.wikipedia.org/wiki/%D0%9C%D1%83%D0%B7%D1%8B%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%82%D0%B5%D0%BC%D0%BF), [тональности](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C) и др., с точной привязкой во времени. Стандартный MIDI-файл – это специально разработанный формат файлов, предназначенный для хранения данных исполняемых секвенсором, [секвенсор](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BA%D0%B2%D0%B5%D0%BD%D1%81%D0%BE%D1%80) может быть, как программой для компьютера, так и аппаратным модулем. Через MIDI интерфейс передается информация о действиях, выполняемых на музыкальном устройстве – к примеру, нажатие клавиши. Это содержит информацию о двух параметрах: номере нажатой клавиши и силе нажатия на нее. Структура MIDI-файла изображена на рисунке 1.6.

Пусть MIDI-карта управляет несколькими устройствами, и каждое устройство исполняет свою часть музыкального произведения, тогда появляется необходимость разделить поток MIDI-сообщений, чтобы каждое устройство могло принимать только свои сообщения. Эта проблема в MIDI решена с помощью каналов. Предполагается, что существует 16 MIDI-каналов, и каждое сообщение может идти только по одному из них. При этом каждое устройство настраивается на прием сообщений, приходящих только по одному из каналов. Поток MIDI-информации передается побайтно. Для контроля за состоянием линии в начале каждого байта передается стартовый бит. Любой байт, передаваемый по MIDI, является байтом либо статуса, либо значения. Статусный байт всегда первый в MIDI-сообщении, он определяет его тип и номер MIDI-канала. В каждом MIDI-сообщении содержится только один статусный байт. Байты значения содержат параметры, необходимые для данного типа MIDI-сообщения. При разработке формата предусмотрели чтобы любой секвенсор мог записывать и читать файл таким образом, чтобы, с одной стороны, не потерялись его данные, а с другой стороны, чтобы файл получился достаточно гибким и приложения могли сохранять в него свою специфическую информацию, понятную только им, но не понятную другим программам-приложениям, причём при загрузке файлов MIDI неизвестная другим программам-приложениям информация не приводит к неполадкам, а просто игнорируется. В этом смысле формат файлов MIDI можно сравнить с форматами текстовой информации. Различные программы-секвенсоры способны читать MIDI-файлы, подобно тому, как различные текстовые редакторы читают [ASCII](https://ru.wikipedia.org/wiki/ASCII)-файлы, которые могут содержать вспомогательную информацию, понятную лишь данному редактору [4].



Рисунок 1.6 – Структура MIDI-файла

Из аналогов стоит отметить программно-аппаратный комплекс Moppy, он включает в себя десктопное приложение, написанное на языке программирования Java, а также аппаратную часть, включающую в себя драйвер, реализованный на плате с микроконтроллером Arduino [5].

Из недостатков рассматриваемого решения необходимо отметить:

* отсутствие в программной части, горячего подключения аппаратной части;
* отсутствие возможности увеличения количества накопителей на гибких магнитных дисках, при количестве последних больше, чем 8, появляется необходимость подключения нового дополнительного драйвера;
* сложная установка на устройства с операционной системой Windows, из-за необходимости установки JDK и драйверов;
* отсутствует возможность воспроизводить звуковые файлы, кроме файлов формата MIDI.

Из преимуществ данного решения стоит выделить возможность подключения MIDI-OUT устройств.

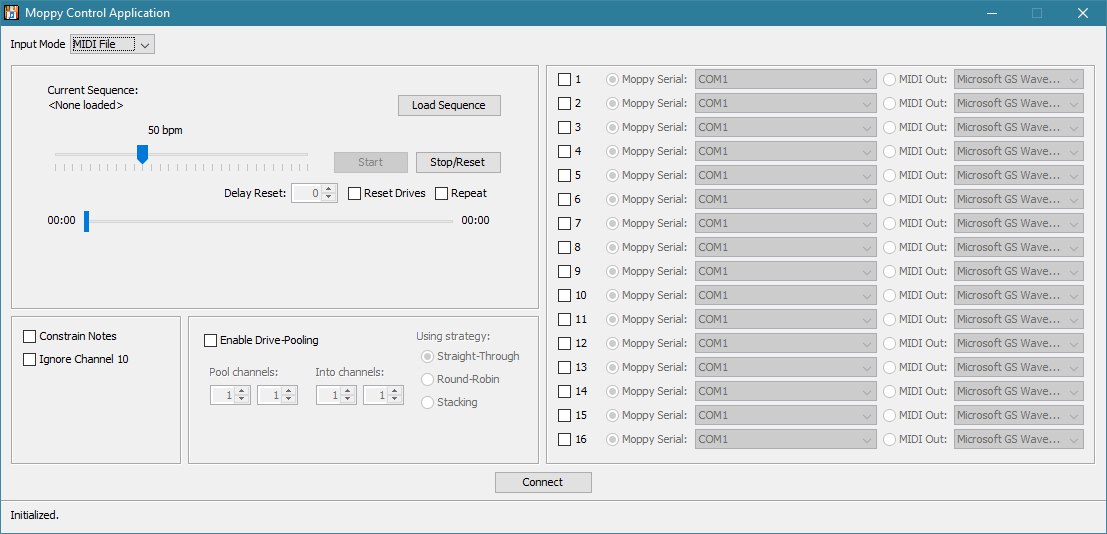


Рисунок 1.7 – Скриншот приложения Moppy

На рисунке 1.7 изображен скриншот главной страницы приложения Moppy.

Следующим аналогом является программно-аппаратный комплекс разработанное участником сообщества Хабрахабр. Данный программно-аппаратный комплекс включает в себя, программу, написанную на языке программирования C++, а также аппаратную часть, реализованную на микроконтроллере Atmega8a, и имеющую беспроводной интерфейс [6].

Из недостатков данного решения можно отметить:

* отсутствие пользовательского интерфейса;
* необходимость заранее подготавливать файл для воспроизведения, отсутствие автоматической генерации управляющих сигналов;
* отсутствие модульности, что не дает нам заменить один из элементов системы;
* отсутствие возможности увеличения количества накопителей на гибких магнитных дисках, при количестве последних больше, чем 8, появляется необходимость подключения нового дополнительного драйвера.

Из преимуществ данного решения можно отметить:

* беспроводной интерфейс подключения аппаратной части проекта;
* дешевизна решения, благодаря использования микроконтроллера Atmega8a.

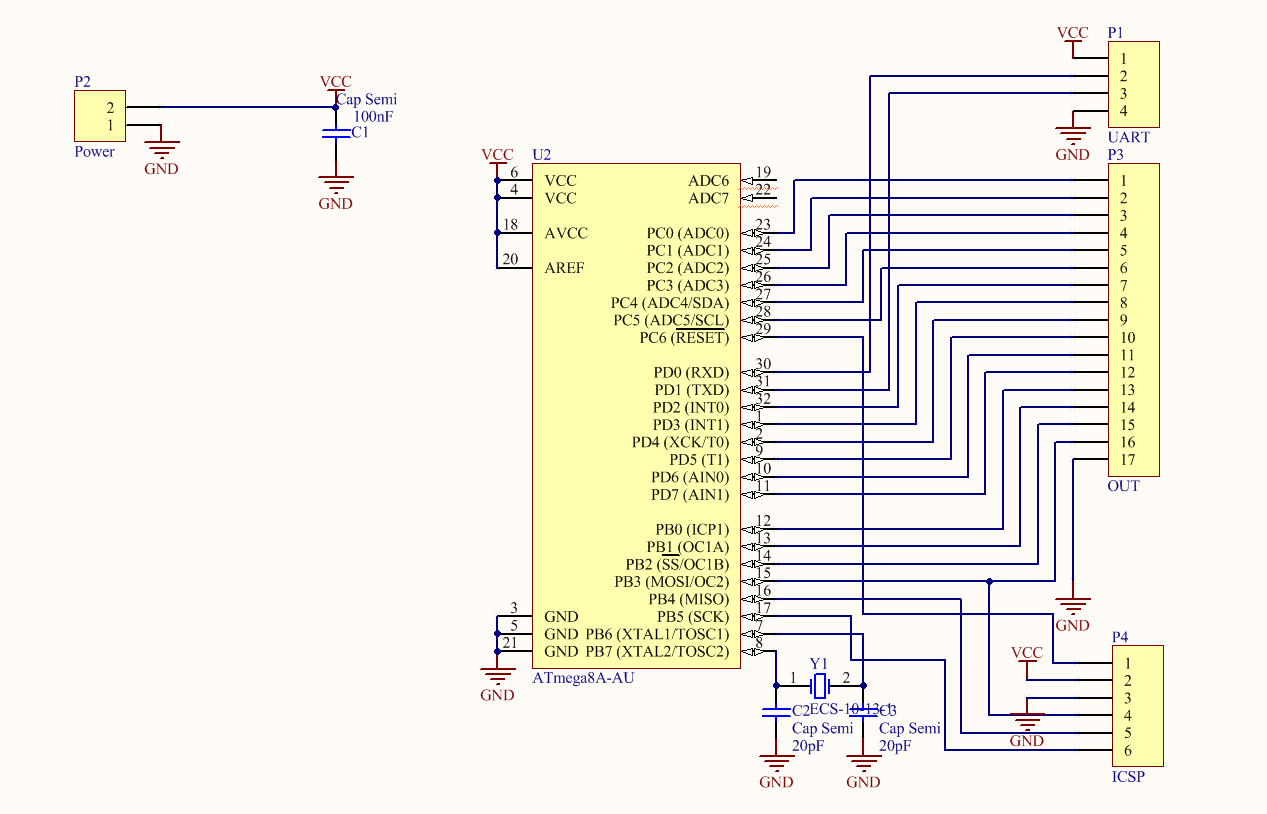


Рисунок 1.8 ­– Схема аппаратной части комплекса

На рисунке 1.8 показана аппаратная часть аналога дипломного проекта.

# **2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

## **2.1** Выбор аппаратной платформы

Задачей аппаратной части дипломного проекта является: проектирование платы расширения и программирование микроконтроллера для управления устройствами ввода-вывода.

Микроконтроллер в данной дипломной работе будет выполнять роль драйвера, а значит, что он должен легко сопрягаться с компьютером, при этом иметь небольшую стоимость. Проанализировав ситуацию была выбрана плата с микроконтроллер торговой марки Arduino, так-как данные платы с микроконтроллером легко расширяются благодаря большому количеству модулей, среди них модули экрана, клавиатуры, Wi-Fi, Bluetooth, Ethernet. А также из коробки имеет возможность сопряжения с компьютером благодаря microUSB.

## **2.2** Выбор целевой платформы и средств разработки

Задачей программной части дипломного проекта является проектирование приложения для обработки и хранения звуковых файлов, а также методов передачи команд на микроконтроллер для управления им.

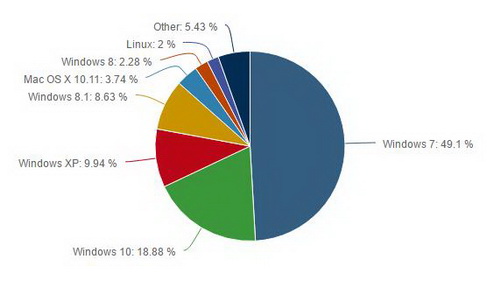


Рисунок 2.1 – Диаграмма разбиения рынка операционных систем

Исходя из этих условий и нужно выбирать подходящую операционную систему и инструменты разработки приложения. Так как в первую очередь приложение будет проектироваться для обычных пользователей, имеет смысл выбрать наиболее популярную операционную систему. На данный момент господствующее место на рынке операционных систем занимает компания Microsoft. Целевыми операционными система для дипломного проекта были выбраны Windows 8, Windows 8.1, Windows 10, т.к. вместе они занимают более 30% рынка операционных систем (см. рисунок 2.1).

Проанализировав основные операции, которые будет выполнять проектируемое приложение, а именно: взаимодействие с файловой системой, работа с базой данных, реализация алгоритмов дискретизации, взаимодействие с контроллером Arduino, предоставление пользователям стандартного Windows интерфейса – было решено выбрать язык программирования C# и технология UWP.

Язык программирования C# является строго типизированным, объектно-ориентированным, безопасным языком программирования, выполняемым в среде .NET Framework. В настоящий момент на нем пишутся самые различные приложения: десктопные приложения, веб-порталов, веб-сервисы. C# поддерживает полиморфизм, наследование, перегрузку операторов, делегатов, статическую типизацию. И в C# активно развивают, и каждой новой версией добавляют новые функции, как, например, лямбды, динамическое связывание, асинхронные методы [7].

Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире. Фреймворк .NET представляет мощную платформу для создания приложений. Можно выделить следующие ее основные черты:

* поддержка нескольких языков. Основой платформы является Common Language Runtime (общеязыковая среда исполнения), благодаря чему .NET поддерживает несколько языков, например, C++, F#;
* кроссплатформенность. Фреймворк .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент поддерживается на большинстве современных ОС Windows, а благодаря проекту Mono можно создавать приложения, которые будут работать и на других ОС семейства Linux;
* мощная библиотека классов. .NET представляет единую для всех поддерживаемых языков библиотеку классов;
* разнообразие технологий. Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET. Для построения сложных веб-приложений – технология ASP.NET. Для построения графических приложений с богатым насыщенным интерфейсом - технология UWP.

Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от С++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память [8].

Для построения пользовательского интерфейса в дипломном проекте используется технология UWP. UWP относительно молодая технология, она была представлена вместе с Windows 10. В основе приложений UWP заложена идея, что пользователя желают использовать любые устройства для продуктивного и удобного выполнения задачи. Microsoft предоставляет разработчику приложений на технологии UWP один набор API, для разработки на всех Windows 10 устройствах, а также упрощенную поддержку различных размеров экрана устройства и адаптивную модель взаимодействия с устройством, а именно: мышь, клавиатура, игровой контроллер, прикосновение пальцем [9].

Учитывая интенсивную работу с данными, нужно выбрать подходящую базу данных. База данных – совокупность данных, организованных в соответствии с концептуальной структурой, описывающей характеристики этих данных и взаимоотношения между ними, причём такое собрание данных, которое поддерживает одну или более областей применения. Традиционно на рынке разработки программного обеспечения лидерами являются SQL базы данных, основанные на реляционном подходе. SQL – формальный непроцедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных. Именно из-за широкой распространённости данного вида баз данных и доступности открытой документации, была выбрана Transact-SQL. Transact-SQL ­– процедурное расширением языка SQL, разработанное компанией Microsoft. Добавляет к языку SQL функции для обработки дат, строк, управляющие операторы и локальные и глобальные переменные [10].

Невозможно представить современную разработку программного обеспечения без использования системы контроля версий. Система контроля версий (СКВ) – это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. В качестве системы контроля версий была выбрана система Git. Это свободная, открытая система контроля версий, которая используется разработчиками всего мира в процессе разработки программных средств. Данная система имеет обширную документацию и интегрируется во множество сервисов, облегчающих разработку программного обеспечения. В дипломном проекте использовался GitHub. GitHub – крупнейший [веб-сервис](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B8%D1%81) для хостинга [IT](https://ru.wikipedia.org/wiki/IT)-проектов и их совместной разработки. Основан на системе контроля версий [Git](https://ru.wikipedia.org/wiki/Git) и разработан на [Ruby on Rails](https://ru.wikipedia.org/wiki/Ruby_on_Rails) и [Erlang](https://ru.wikipedia.org/wiki/Erlang) компанией GitHub. Благодаря своей популярности GitHub имеет большое количество разнообразных клиентов и модулей для Visual Studio.

## **2.3** Планирование логических модулей

Весь разрабатываемый проект разбивается на отдельные модули. Данный подход позволяет добиться гибкой структуры программы. Хорошо продуманная модель абстракций каждого блока позволяет менять внутреннюю структуру блока, сохраняя оговоренные интерфейсы, что в итоге никоим образом не отразится на функциональности других модулей. Разбиение на модули происходит исходя из задач, решаемых программных обеспечением:

* работа с базой данных;
* реализация алгоритмов дискретизации;
* реализация бизнес-логики;
* реализацию прослушивания подключенных устройств;
* генерация команд для управления драйвером;
* взаимодействие с драйвером;
* предоставление пользователям стандартного Windows интерфейса;
* взаимодействие с файловой системой.

Каждый логический модуль решает одну или несколько вышеперечисленных задач. Кроме того, каждый модуль программного обеспечения взаимодействует с остальными модулями, что обеспечивает работоспособность всего приложения в целом. Связь реализуется при помощи интерфейсов и IoC-контейнера.  IoC-контейнер – это [компоновщик](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA), который собирает не [объектные файлы](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C), а объекты [ООП](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) ([экземпляры класса](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))) во [время исполнения](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%BD%D1%82%D0%B0%D0%B9%D0%BC) программы. IoC-контейнер реализован в виде паттерна фабрики. Выбрав данную структуру проекта, мы с легкостью сможем заменять ключевые модули приложения на другие, делая это во время выполнения программы, используя для этого IoC-контейнер.

### **2.3.1** Модуль взаимодействия с базой данных

Данный модуль представляет собой абстракцию для удобного взаимодействия языка программирования с базой данных. Именно он выполняет всю низкоуровневую работу с базой данных, предоставляя программисту простые абстрактные интерфейсы для взаимодействия, данный модуль взаимодействует с модулем бизнес-логики и модулем алгоритмов дискретизации.

### **2.3.2** Модуль взаимодействия с файловой системой

Данный модуль представляет собой набор клиентского программного обеспечения, которое взаимодействует непосредственно с файловой системой, данный модуль должен адаптивно работать с различными устройствами, данный модуль предоставляет отображение файловой системы модулю пользовательского интерфейса, а также предоставляет файлы модулю бизнес-логики.

### **2.3.3** Модуль алгоритмов дискретизации

Занимается обработкой данных, полученных от модуля взаимодействия с базой данных. Основной задачей данного модуля является дискретизация полученные данные и передача полученный результат в модуль взаимодействия с базой данных для сохранения результатов дискретизации.

### **2.3.4** Модуль реализации бизнес-логики

Данный модуль отвечает непосредственно за процессы бизнес-логики в приложении. Также он отвечает за взаимодействие между базой данных и пользовательским интерфейсом, а также он принимает данные из модуля поиска подключенных устройств, и передает данные в модуль генерации команд для управления драйвером.

### **2.3.5** Модуль прослушивания подключенных устройств

С помощью данного модуля происходит поиск подключенных устройств, целью данного модуля является возможность горячего подключения аппаратной части проекта.

### **2.3.6** Модуль генерации команд, для управления драйвером

Данный модуль представляет собой абстракцию, для удобного управления драйвером. Он выполняет низкоуровневую работу по управлению драйвером.

### **2.3.7** Модуль взаимодействия с драйвером

Данный модуль представляет собой абстракцию, для удобного взаимодействия между драйвером и устройством. Он выполняет низкоуровневую работу по передаче данных и команд.

### **2.3.8** Модуль взаимодействия с драйвером

С помощью данного модуля происходит всё взаимодействие пользователя с программной частью комплекса. В частности, этот модуль отвечает за визуализацию процесса воспроизведения, обработки и загрузки музыкальных композиций.

# **3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе будет подробно описано функционирование всех блоков программно-аппаратного комплекса.

В предыдущем разделе были выбраны инструменты разработки программно-аппаратного комплекса. Было решено использовать .Net фреймворк и аппаратно-программный инструмент Arduino как средства, позволяющие наиболее эффективным образом решить поставленную задачу. Особенностью разработки программного средства с использованием каких-либо фреймворков является то, что на разработчика накладываются определённые архитектурные рамки и дополнительные требования. Несмотря на это, дополнительные требования обычно положительным образом сказываются на качестве кода и лёгкости дальнейшей поддержки данного кода, а архитектурные ограничения позволяют создавать приложения по принципам, считающимися "лучшими практиками" в среде программистов.

Учитывая, что программной-аппаратный комплекс будет создаваться с использованием .Net фреймворка, то для более глубокого понимания работы программной-аппаратный комплекса, для начала, рассмотрим подробно архитектуру и основные принципы данного фреймворка.

### **3.1** Описание принципов работы .Net фреймворк

Сам по себе .Net фреймворк является каркасом для разработки приложений, предоставляя дополнительные средства для упрощения обработки данных, обработки пользовательских запросов, предоставления пользователю графического представления, работы с асинхронностью. Разберем основные принципы .Net фреймворк.

CLR (Common Language Runtime) – общеязыковая исполняющая среда. Она обеспечивает независимость и интеграцию языков программирования и позволяет различным модулям приложения созданных на разных языках равноправно взаимодействовать между собой. Данный механизм, благодаря компилятору определенного языка программирования, переводит код написанный на определенном языке в IL-код. Среди поддерживаемых языков программирования уже есть: C#, VB .Net, Fortran, F# и множество других языков программирования. На рисунке 3.1 показан основной принцип работы CLR. Чтобы включить в среде выполнения предоставление служб управляемому коду, языковые компиляторы должны предоставлять метаданные с описанием типов, членов и ссылок в коде. Метаданные хранятся вместе с кодом. Они содержатся в каждом загружаемом переносимом исполняемом файле среды CLR. Метаданные в среде выполнения используются для поиска и загрузки классов, размещения экземпляров в памяти, разрешения имен при вызове методов, создания машинного кода, обеспечения безопасности и установки границ контекста времени выполнения.

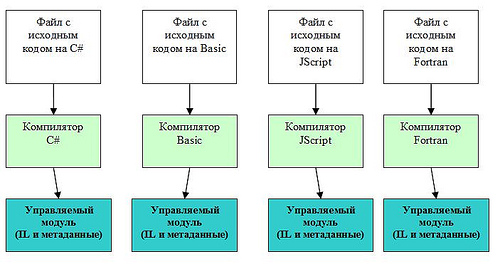


Рисунок 3.1 – Принцип работы CLR

IL (Intermediate Language) – процессора-независимый машинный язык программирования, разработанный компанией Microsoft. В отличии от других машинных языков IL позволяет работать с объектами, а также имеет команды для работы с виртуальными методами и исключения, манипуляциями с массивами. IL можно назвать объекто-ориентированным машинным языком программирования. Процесс генерации и выполнения IL-кода показан на рисунке 3.2.

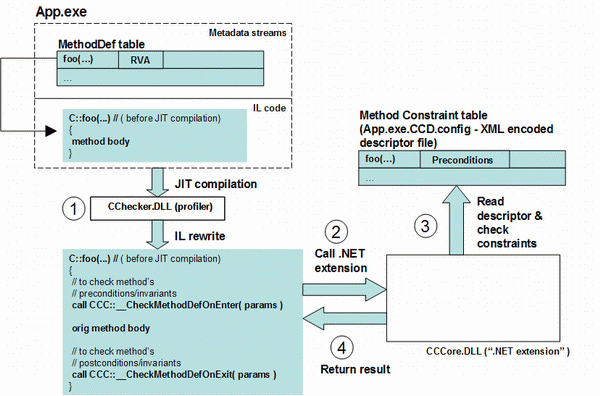


Рисунок 3.2 – Процесс генерации и выполнения IL-кода

Для ознакомления с IL-кодом необходимо воспользоваться утилитой IL DASM. Скриншот утилиты IL DASM показан на рисунке 3.3.

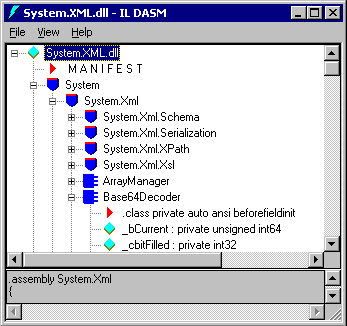


Рисунок 3.3 – Скриншот утилиты IL DASM

CLS (Common Language Specification) – спецификации выпущенная компанией Майкрософт. Данная спецификация описывает минимальный набор возможностей, которые должны реализовать производители компиляторов, чтобы их продукты работали в CLR. В данную спецификацию включаются: ключевые слова, символы и регистры, простые типы, операторы преобразования, а также правила перегрузки типов, модификаторы доступа и

правила обработки событий.

Одна из основных функция .Net фреймворка это асинхронность. Асинхронность подразумевает, что операция может быть выполнена каким-либо классом из вне. Основное свойство таких операций в том, что начало такой операции требует значительно меньшего времени, чем основная работа. Что позволяет выполнять множество асинхронных операций одновременно. На рисунке 3.5 показано преимущество асинхронности над обычным выполнением кода.

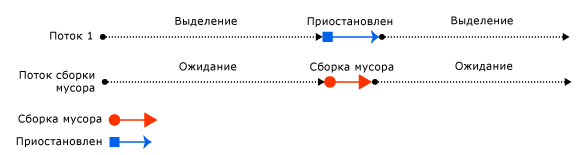
****

Рисунок 3.4 – Сборка мусора в .NET

Сборщик мусора является одним из преимуществ .Net фреймворка. Рассмотри работу сборщика мусора на примере языка программирования C#. При инициализации объекта ключевым словом new, данный объект попадает в управляемую кучу. Когда пропадает необходимость в данном объекте сборщик мусора автоматически удалит его.

При попытке обнаружить недостижимый код объекты CLR-среды не проверяют буквально каждый находящийся в куче объект.  
Для оптимизации процесса каждый объект в куче относится к определённому “поколению”. Смысл в применении поколений выглядит довольно просто: Чем дольше объект находится в куче, тем выше вероятность того, что он там будет оставаться.

Однако есть и недостатки в данном решении, а именно, поток приложение приостанавливает работу на момент, когда совершается сборка мусора. Данная остановка может повлиять на быстродействие приложения. На рисунке 3.4 показан процесс приостановки потока приложения.

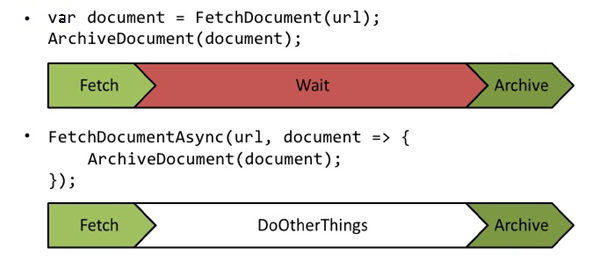


Рисунок 3.5 – Процесс асинхронности в .NET

### **3.2** Описание работы модуля взаимодействия с базой данных

Множество приложений, как правило, в качестве основного хранилища данных используют базы данных. В разрабатываемом программном-аппаратном комплексе также в качестве основного хранилища задействована реляционная база данных Transact SQL.

Все манипуляции с базой данных происходят с помощью модуля взаимодействия с базой данных. Данный модуль предоставляет интерфейс для работы с базами данных. Работа данного модуля выполняется целиком за счёт средств Entity Framework. В свою очередь Entity Framework является надстройкой для ADO.NET. Перед тем как разбираться с Entity Framework необходимо понять, как работает ADO.NET.

### **3.2.1** ADO.NET

ADO.NET – это набор классов, предоставляющий программистам службу для доступа к данным приложения. ADO.NET предоставляет доступ к различным источникам данных, таким как: реляционные базы данных, XML-данным и другим.

ADO.NET разделят доступ к данным и обработку данных на дискретные компоненты, которые могут использоваться отдельно или совместно. В ADO.NET используется многоуровневая архитектура, которая обращается вокруг небольшого числа ключевых концепций, таких как поставщики данных.

Поставщик данных– это набор классов ADO.NET, которые позволяют получать доступ к определенной базе данных, выполнять команды SQL и извлекать данные.

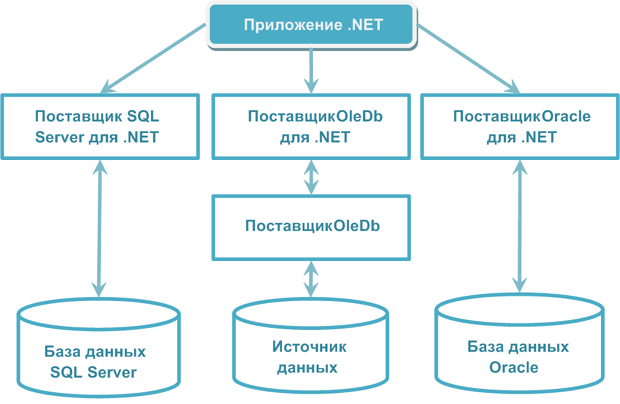


Рисунок 3.4 – Поставщики ADO.NET

На рисунке 3.4 изображены поставщики данных ADO.NET.

В первом приближении поставщик данных можно рассматривать как набор типов, определенных в данном пространстве имен, который предназначен для взаимодействия с конкретным источником данных. Однако независимо от используемого поставщика данных, каждый из них определяет набор классов, обеспечивающих основную функциональность. В таблице 3.1 приведены некоторые общие основные объекты, их базовые классы и основные интерфейсы, которые они реализуют.

В ADO.NET термин "объект подключения" на самом деле относится к конкретному типу, порожденному от DbConnection; объекта подключения "вообще" нет. То же можно сказать и об "объекте команды", "объекте адаптера данных" и т.д. По соглашению имена объектов в конкретном поставщике данных имеют префиксы соответствующей СУБД (например, SqlConnection, OracleConnection, SqlDataReader и т.д.).

Таблица 3.1 – Основные объекты поставщиков данных ADO.NET

|  |  |  |  |
| --- | --- | --- | --- |
| Тип объекта | Базовый класс | Соответствующие интерфейсы | Назначение |
| Connection | DbConnection | IDbConnection | Позволяет подключаться к хранилищу данных и отключаться от него. Кроме того, объекты подключения обеспечивают доступ к соответствующим объектам транзакций. |
| Command | DbCommand | IDbCommand | Представляет SQL-запрос или хранимую процедуру. Кроме того, объекты команд предоставляют доступ к объекту чтения данных конкретного поставщика данных. |
| DataReader | DbDataReader | IDataReader, IDataRecord | Предоставляет доступ к данным только для чтения в прямом направлении с помощью курсора на стороне сервера. |
| DataAdapter | DbDataAdapter | IDataAdapter, IDbDataAdapter | Пересылает наборы данных из хранилища данных к вызывающему процессу и обратно. Адаптеры данных содержат подключение и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации в хранилище данных. |
| Parameter | DbParameter | IDataParameter, IDbDataParameter | Представляет именованный параметр в параметризованном запросе. |
| Transaction | DbTransaction | IDbTransaction | Инкапсулирует транзакцию в базе данных. |

### **3.2.2** Entity Framework

Entity Framework – является ORM решением для .Net. Данный фреймворк позволяет работать с базой данных при помощи LINQ либо Entity SQL.  Если традиционные средства ADO.NET позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Центральным понятием в концепции Entity Framework является сущность. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами. Для описания сущности используются POCO (Plain Old CLR Object) объекты. Для описания полей таблицы базы данных используются свойства POCO объектов. Для того чтобы Entity Framework мог получить доступ к базе данных, в системе должен быть установлен соответствующий провайдер. На рисунке 3.5 показана схема взаимодействия Entity Framework с приложением.

Entity Framework позволяет описывать со связями один к одному, один ко многим и многие ко многим. А также Entity Framework поддерживает такие технологии как миграции, отложенная загрузка и прямая загрузка.

Миграции – механизм, который позволяет менять внутреннюю структуру базы данных, что намного упрощает задачу программиста, и позволяет не писать вручную SQL запросы для изменения структуры базы. Каждую миграцию можно рассматривать как изменение базы данных. Схема изначально ничего не содержит, а каждая миграция изменяет ее, добавляя или убирая таблицы, столбцы или записи. Все файлы миграции в названии имеют временную метку, таким образом разработчик имеет полноценную историю изменения структуры базы данных с начала разработки сервиса.

Отложенная загрузказаключается в том, что Entity Framework автоматически загружает данные, при этом не загружая связанные данные. Когда потребуются связанные данные Entity Framework создаст еще один запрос к базе данных.

Entity Framework применяет отложенную загрузку, используя динамические прокси-объекты. Когда Entity Framework возвращает результаты запроса, он создает экземпляры классов и заполняет их данными, которые были возвращены из базы данных. Entity Framework имеет возможность динамически создавать новый тип во время выполнения, производный от класса модели POCO. Этот новый класс выступает в качестве прокси-объекта для класса POCO и называется динамическим прокси-объектом. Он будет переопределять навигационные свойства вашего класса POCO и включать в себя некоторую дополнительную логику для извлечения данных из базы данных, когда вызывается навигационное свойство.

Прямая загрузка данныхпозволяет указать в запросе какие связанные данные нужно загрузить при выполнении запроса. Благодаря этому, когда в коде будет использоваться ссылку на связанную таблицу через навигационное свойство, SQL-запрос не будет направляться в базу данных, т.к. связанные данные уже будут загружены при первом запросе. В Entity Framework для этих целей используется метод Include(), которому передается делегат, в котором можно указать навигационное свойство, по которому данные должны загружаться при первом запросе. Этот метод является расширяющим для IQueryable.

Каскадное удаление данных позволяет удалять связанные данные из зависимой таблицы, при удалении данных из основной таблицы.

Однако для того, чтобы можно было использовать все эти преимущество, нужно следовать определённым соглашениям фреймворка. Так же желательно соблюдать правила по именованию моделей.

В Entity Framework существует несколько подходов к созданию моделей и базы данных:

1. Database First – данный метод позволяет нам создавать модель, из уже существующей базы данных, хорошее решения, когда разработчик уже имеет готовую базу данных.
2. Model First – позволяет нам создавать модель базы данных через визуальный редактор. Суть данного подхода состоит в том, что сначала делается модель, а потом по ней создается база данных, данный метод идеально подходит для архитекторов.
3. Code First – в данном подходе мы имеем возможность создать модель базы данных при помощи кода. Этот подход очень прост и удобен. Но он также и очень гибкий.



Рисунок 3.5 – Принцип работы Entity Framework

Для настройки типов столбцов и таблиц можно использовать аннотации или Fluent API. Fluent API позволяет нам настроить ограничения на таблицу и колонки.

Далее подробным образом описываются все основные структуры данных, задействованные в работе программно-аппаратного комплекса, их поля и основное предназначение.

**3.2.3** Таблица Music

Данная таблица хранит информацию обо всех добавленных в приложение композиций.

Поля:

* uid – уникальный идентификатор композиции;
* Name – наименование композиции;
* Extention – расширение файла композиции;
* Date – дата добавления композиции в приложение;
* Content – байтовое представление загруженной композиции;
* ConvertedMusic – ссылка на композицию над которой производилась конвертация.

### **3.2.4** Таблица СonvertedMusic

Данная таблица хранит информацию обо всех конвертациях файлов в приложении.

Поля:

* uid – уникальный идентификатор конвертированной композиции;
* ConvertedDate – дата конвертации композиции в приложении.

### **3.2.5** Таблица Note

Данная таблица предназначена для хранения информации о ноте в композиции.

Поля:

* uid – уникальный идентификатор ноты композиции;
* RealTime – момент времени относительно начала композиции, в который данная нота воспроизводится;
* NoteNumber – номер ноты;
* Channel – канал, в котором нота воспроизводится;
* CommandCode – тип нотной команды.

### **3.2.6** Таблица Albums

Данная таблица хранит информацию обо всех альбомах в приложении.

Поля:

* uid – уникальный идентификатор конвертированной композиции;
* creationDate – дата создания альбома в приложении;
* name – наименование альбома.

### **3.2.7** Таблица Music\_Albums

Данная таблица хранит связь многое ко многому между композициями и альбомами.

Поля:

* musicUid – уникальный идентификатор композиции;
* albumUid – уникальный идентификатор альбома.

### **3.3** Работа модуля дискретизации

Данный модуль предназначен для обработки поступивших в приложения музыкальных файлов и преобразования их в набор нот. Данный модуль крайне важен, ведь от качества дискретизации зависит насколько хорошо будет воспроизводиться композиция на устройствах ввода-вывода. Для работы данного модуля нам понадобилась библиотека NAudio.

NAudio – это .Net библиотека с открытым исходным кодом. Данная библиотека предназначена для работы с музыкальными форматами, такими как MIDI, MP3, WAV. NAudio находится в разработке с 2002 года.

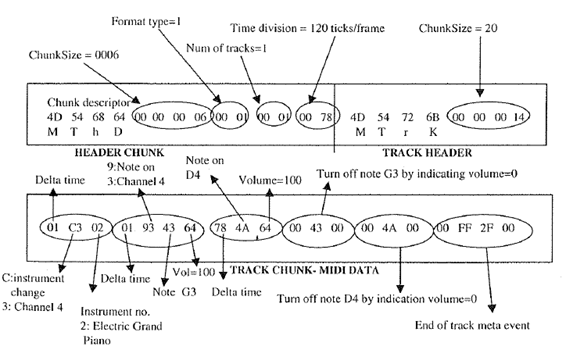


Рисунок 3.6 – Описание формата MIDI-файла

Для качественного воспроизведения звуковых композиций на устройствах ввода-вывода данную композицию необходимо разложить на множество событий. Основными событиями в нашем дипломном проекте стоит считать события включения и выключения ноты. При этом для грамотного включения и выключения ноты нам необходимо знать некоторые составляющие данного события, а именно номер ноты, номер канала, код события и время выполнения события, разберем данные понятия.

Номер ноты – это численное представления ноты, воспроизводимой в композиции. И тут возникает трудность, при разложении MIDI-файлов появляется необходимость устанавливать номер ноты в зависимость от инструмента, который данную ноту воспроизводит. В файлах типа WAV и MP3 данная необходимость пропадает, т.к. мы не берем конкретную ноту из файла, как мы это делаем в MIDI-файлах, а берем звук, воспроизводимый в нужный нам момент. На рисунке 3.6 показан формат MIDI-файла.

Номер канал – показывает на каком из устройств ввода-вывода будет воспроизводиться данная нота. При достаточном количества устройств, мы можем воспроизводить одну ноту на нескольких устройствах, достигая тем самым высокую громкость, однако при недостаточном количестве устройств появляется необходимость объединять некоторые ноты, что сказывается на качестве воспроизводимой композиции.

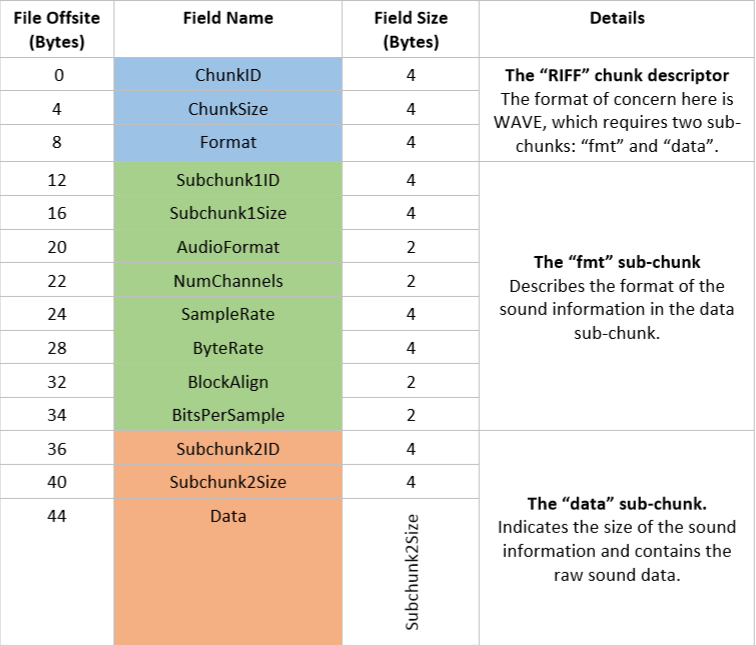


Рисунок 3.7 – Описание формата WAV-файла

Код события – указывает устройству ввода-вывода, что ему необходимо делать с пришедшей информацией. В случае MP3- и WAV- файлов у нас возможно только 2 события, включения и отключения ноты на устройстве. В случае же MIDI-файлов появляются дополнительные события, такие как начало композиции, смена темпа композиции, окончание композиции, смена инструмента и другие.

Время выполнения события – показывает нам, когда данное событие должно произойти. В отличии от WAV- и MP3-файлов, где события происходят через равный промежуток времени, который был выбран нами для дискретизации, в MIDI-файлах мы имеем набор нот, к которым привязан промежуток времени в тиках, показывающий, когда именно данное событие должно произойти. Трудность использования тиков в том, что время, которое обозначает один тик, может изменяться во время проигрывания композиции большое количество раз. Для определения точного времени, которое обозначают тики, нам необходимо использовать события изменения темпа и начальное значения длины четверть ноты.

Так как библиотека NAudio позволяет нам легко преобразовывать MP3-файлы в WAV-файлы, нам достаточно преобразовывать WAV-файла. Структура WAV-файла показана на рисунке 3.7.

Зная вышеперечисленные значения, необходимо подавать их в нужном порядке на устройства ввода-вывода для воспроизведения звуковой композиции.

### **3.4** Работа модуля реализации бизнес-логики

Данный модуль является промежуточным модулем между модулем взаимодействия с базой данных и модулем построения графического интерфейса и служит для удобного преобразования данных.

В основном данный модуль служит для вызова различных классов для преобразования данных, а также передачи полученных данных модулю работы с базой данных. Основной работой данного модуля является обеспечение асинхронного выполнения методов различных модулей, отвечающих за обработку данных. Для правильного выполнения асинхронных операций в .Net-фреймворке присутствуют операторы async и await. Ключевое слово async указывает, что метод или лямбда-выражение являются асинхронными. А оператор await применяется к задаче в асинхронных методах, чтобы приостановить выполнение метода до тех пор, пока эта задача не завершится. При этом выполнение потока, в котором был вызван асинхронный метод, не прерывается.

Асинхронные методы предназначены для неблокирующих операций. Выражение await в асинхронном методе не блокирует текущий поток, в то время как выполняется ожидаемая задача. Вместо этого, выражение регистрирует остальную часть метода как продолжение и возвращает управление вызывающему объекту асинхронного метода. Ключевые слова async и await не вызывают создание дополнительных потоков. Асинхронные методы не требуют многопоточности, поскольку асинхронный метод не выполняется в своем собственном потоке. Он выполняется в текущем контексте синхронизации и использует время в потоке, только когда метод активен. [Task.Run](https://msdn.microsoft.com/ru-ru/library/system.threading.tasks.task.run(v=vs.110).aspx) можно использовать для перемещения работы, ограниченной производительностью процессора, в фоновый поток, но фоновый поток не помогает с процессом, который просто ожидает, когда результаты станут доступными.

### **3.5** Работа модуля прослушивания подключенных устройств

Данный модуль необходим для PnP-подключения устройств. PnP (Plug And Play) – это технология, для быстрого подключения и конфигурирования устройств. Для работы данного модуля используется класс DeviceWatcher. Данный класс позволяет нам подписаться на события подключения и отключения устройств. На рисунке 3.8 отображены взаимодействий событий класса DeviceWatcher.

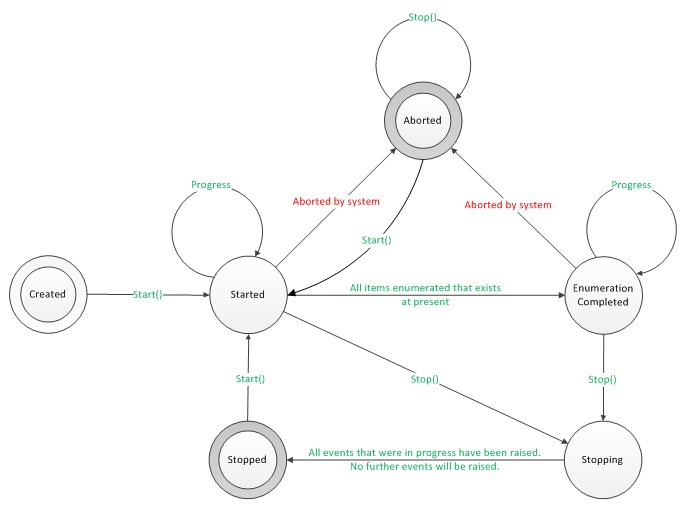


Рисунок 3.8 – Принцип событий класса DeviceWatcher

Рассмотри события, которые поддерживаются классом DeviceWatcher:

1. Added – событие срабатывает в момент, когда DeviceWatcher обнаруживает новое устройство подключенное к системе.
2. Removed – событие срабатывает в момент, когда устройство отключается от системы.
3. Updated – событие срабатывает, когда устройство обновляет информацию о себе.
4. Stopped – событие срабатывает, когда мы прекращаем прослушивание устройств.

Для корректной работы класса DeviceWatcher необходимо реализовать фильтр, который будет прослушивать только необходимые для нас устройства.

**3.6** Модуль генерации команд, для управления

Для упрощенного управления драйвером из приложения мы использовали библиотеку Windows Remote Arduino.

Windows Remote Arduino – это .Net библиотека с открытым исходным кодом, которая позволяет любому Windows 10 устройству управлять Arduino при помощи Bluetooth, USB, WiFi, или Ethernet соединения.

Windows Remote Arduino позволяет управлять такими функциями Arduino, как:

* Управление аналоговыми и цифровыми входами/выходами;
* Передача данных между устройствами посредством I2C;
* Позволяет нам создавать собственный набор команд.

API данной библиотеки придерживается команд, которые уже используются в Arduino. Кроме API необходим протокол, который бы облегчил бы нам связь между Windows 10 и Arduino. Данным протоколом является протокол Firmata. Библиотека Firmata уже включена в интегрированную среду разработки Arduino по умолчанию. Данная библиотека прекрасно интегрируется с DeviceWatcher, позволяя нам управлять Arduino-устройствами подключенных к Windows 10 устройствам.

Рассмотрим подробнее технологию Windows Remote Arduino.

**3.6.1** Архитектура технологии Windows Remote Arduino

Windows Remote Arduino имеет трехуровневую архитектуру, что позволяет нам разграничить некоторые функции, и установить между ними слабую связь. На рисунке 3.9 изображена архитектура данной библиотеки.

Нижним уровнем данной архитектуры является физический уровень. Данный уровень предназначен для обмена необработанных данных между Windows 10 и устройством Arduino.

Вторым уровнем данной архитектуры является уровень протокола Firmata, который декодирует необработанные данные в осмысленные сообщения.

Третьим уровнем является API, абстрагирующим все сообщения протокола и позволяет дистанционно управлять устройства Arduino.

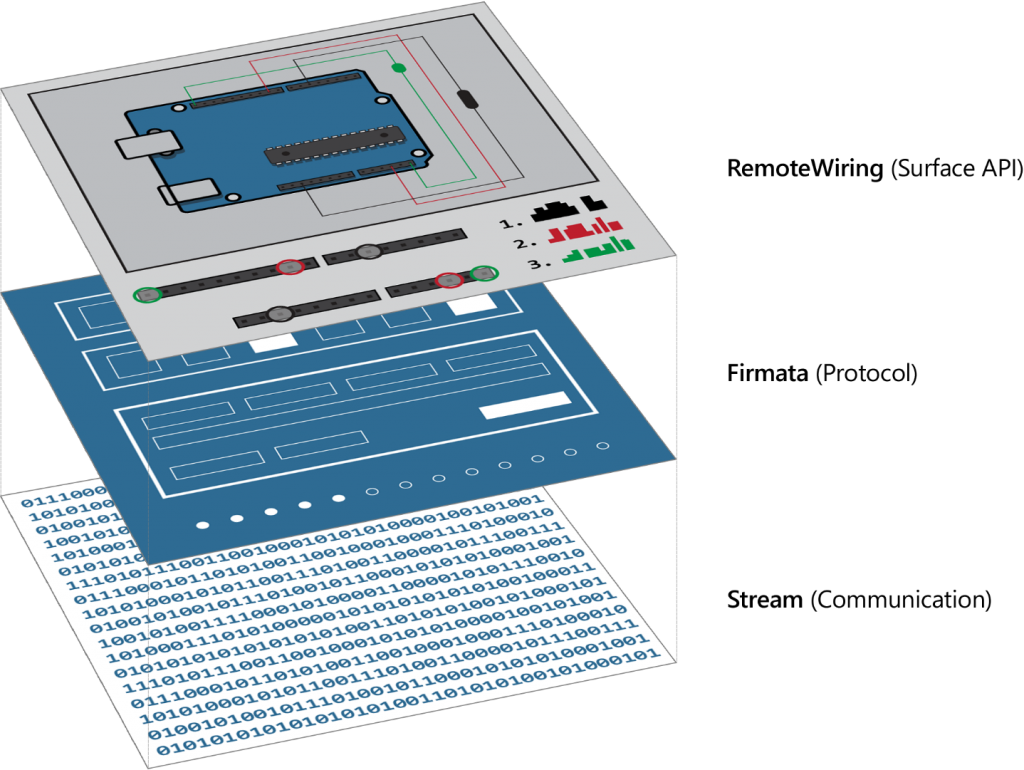


Рисунок 3.9 – Архитектура Windows Remote Arduino

### **3.7** Модуль взаимодействия с драйвером

Данный модуль предназначен для асинхронного обмена данными и командами между устройствами посредством библиотеки Windows Remote Arduino. Данный модуль, пользуясь модулем генерации команд для управления драйвером, генерирует последовательность команд для управления драйвером. Данной модуль способен генерировать команды для драйвера без внутренней памяти, в этом случае ответственность за управлением драйвером полностью на себя берет данный модуль, в его обязанности входит поддержания необходимого такта выполнения команд, проверка наличия связи с драйвером, диспетчеризация устройств ввода-вывода. В случае же если драйвер владеет внутренней памятью, то модуль генерирует набор команд для сохранения композиций на драйвере, в этом случае данный модуль отвечает только за синхронизацию между компьютером и драйвером, а поддержание нужного такта и диспетчеризацией занимается драйвер. Стоит отметить, что модуль не зависит от способа подключения драйвера к компьютеру, синхронизация происходит во всех случаях одинаково, благодаря Windows Remote Arduino.

Для синхронизации и поддержания необходимого такта команд используется класс Timer, данный класс находится в пространстве имен System.Threading. Данный класс предоставляет механизм генерации событий в пуле потока с заданным интервалом времени. Когда мы создаем таймер, мы можем указать время ожидания перед первым выполнением метода и время ожидания между последующими выполнениями. Класс System.Threading.Timer имеет такое же разрешение, как системные часы. Это означает, что, если период меньше разрешения системных часов, делегат System.Threading.TimerCallback будет выполняться с интервалами, определяемыми разрешением системных часов, что составляет приблизительно 15 миллисекунд в системах Windows 8 и Windows 10. Для остановки работы таймера необходимо вызывать метод Dispose.

### **3.8** Модуль построения пользовательского интерфейса

Данный модуль предназначен для построения пользовательского интерфейса, который предназначен для удобного отображения и сбора информации.

Для построение пользовательского интерфейса в приложении использовалась платформа UWP.

В платформе UWP для упорядочивания и группирования элементов пользовательского интерфейса используются панели. В большинстве панелей макета XAML задаются прикрепленные свойства, благодаря которым дочерние элементы данной панели сообщают родительской собственное расположение. Если у нас имеются панели, вложенные в другие панели, то вложенные свойства элементов пользовательского интерфейса, которые указывают характеристики макета родительского элемента, интерпретируются самой непосредственной родительской панелью.

Важной особенностью пользовательских интерфейсов является их адаптируемость под различные разрешения экранов. В UWP верстка имеет блочную структуру, а это значит, что если блок не помещается по ширине, то он перемещается под соседний блок. Также для адаптивности в UWP имеется специальный механизм прикрепления панелей к специальным точкам, которые располагаются в крайних точках окна приложения, что позволяет нам растягивать панель вместе с окном приложения.

Для позиционирования элементов в окне приложения отвечают таких свойства, как margin и padding. На рисунке 3.10 показаны основные свойства для позиционирования элементов в окне приложения.

Чтобы наше приложение выглядело приятным для конечного пользователя мы использовали стили, использование которых облегчено в UWP. Для улучшения пользовательского интерфейса у разработчиков есть возможность изменять такие свойства элементов, как цвет, анимацию, шрифты, а также фоновые изображения и иконки.

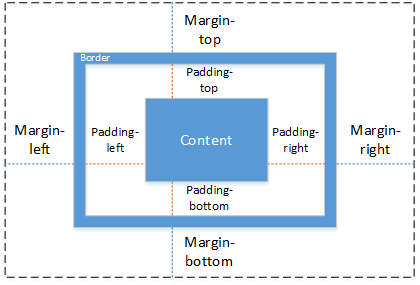


Рисунок 3.10 – Позиционирование элементов пользовательского

интерфейса

Для облегчения разработки пользовательского интерфейса разработчиками платформы UWP были добавлены шаблоны ввода. Шаблоны ввода позволяют разработчику использовать уже готовые решения для ввода простых данных, а также сложных структур. Все шаблоны реализованы в виде XAML-узлов, которые настраиваются при помощи свойств и подузлов данных XAML-узлов. На данный момент на платформе UWP доступно более 45 типов узлов. Кроме стандартных шаблонов ввода у разработчиков есть возможность использовать специальные шаблоны, разработанные сторонними компаниями.

Приложения UWP автоматически обрабатывают широкий спектр интерфейсов и работают на различных устройствах ввода, например, при переносе приложения на мобильную платформу, ввод с сенсорного экрана будет обрабатываться автоматически. Но бывают случаи, когда вы можете оптимизировать свое приложение для определенных типов ввода или устройств. Например, если вы создаете приложение рисования, вы можете настроить способ обработки ввода пером, но таки случаи редки. UWP поддерживает автоматическую обработку большого количества способов ввода, а именно клавиатура, мышь, игровые джойстики и даже голосовой помощник Кортана.

### **3.9** Модуль взаимодействия с файловой системой

Данный модуль предназначен для упрощения работы с файловой системой. Для взаимодействия с файловой системой используется класс FileOpenPicker. Данный класс реализовывается технологией UWP и поддерживает работу со всеми устройствами, на которых установлена Windows 10. На рисунке 3.11 показано окно выбора документа.

При работе с объектами данного класса появляется необходимость настроить данный объект, рассмотрим некоторые конфигурации объектов данного класса:

* FileTypeFilter – данная конфигурация является перечислением, в которое можно добавлять расширения файлов, которые мы можем просматривать;
* SuggestedStartLocation – данная конфигурация, указывает с какой папки у нас открывается отображение файловой системы, и представляет собой значение из перечисления PickerLocationId;
* PickerViewMode – данная конфигурация позволяет нам настроить отображение файлов при выборе, принимает значение из перечисления PickerViewMode.

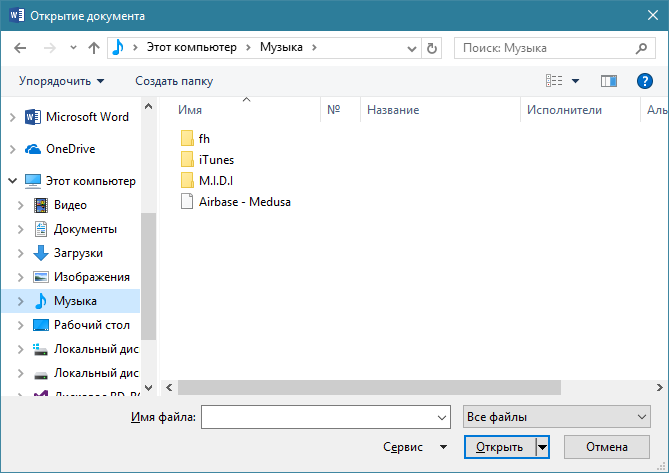


Рисунок 3.11 – Пример окна выбора документ

Класс FileOpenPicker позволяет выбирать как один файл, так и несколько файлов сразу, при этом все операции с файловой системой осуществляются асинхронно.

### **3.10** Принцип инверсии управления

Данный принцип позволяет нам объединять все вышеперечисленные модули, а также предоставляет возможность реализации слабосвязанного кода, зависимости которого проставляются не по реализациям, а по интерфейсам. Данный способ задания связей позволяет нам заменять любую из компонент приложения, без изменений других компонент, к примеру, мы можем заменить модуль пользовательского интерфейса и сделать веб-приложение, при этом не заменяя бизнес-логику и алгоритмы доступа к базам данных. Одной из основных реализаций данного принципа является принцип внедрения зависимостей. На рисунке 3.12 графически показан принцип внедрения зависимостей. Для реализации данного принципа был использован IoC-контейнер Ninject.

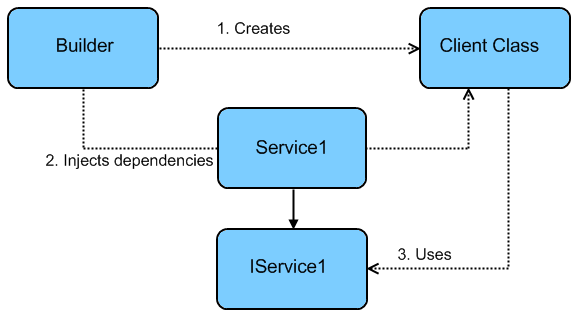


Рисунок 3.12 – Принцип внедрения зависимостей

**3.10.1** Описание IoC-контейнера Ninject

Ninject, в качестве IoC-контейнера, предоставляет нам возможность легкого и быстрого описания зависимостей между интерфейсами и классами. Для этого необходимо задать ядро, в котором будут храниться все необходимые связи. Процесс установки связей между интерфейсами и модулями называется конфигурирования. Во время конфигурирования мы явно задаем связи между интерфейсами и классами, реализующими данный интерфейс. Одним из плюсов использования IoC-контейнера вместо установки статических связей, это то, что у нас появляется возможность заменять основные модули приложения прямо во время выполнения программы, для этого необходимо всего лишь переконфигурировать ядро. Далее мы можем, обращаясь к ядру, получать реализацию необходимого нам интерфейса. В основном IoC-контейнере являются реализацией паттерна Фабрика, мы могли бы использовать его в нашем дипломном проекте, но использование IoC-контейнера Ninject позволило нам сократить код и время на описание зависимостей в приложении. На рисунке 3.13 показан пример использования контейнера Ninject в приложении.

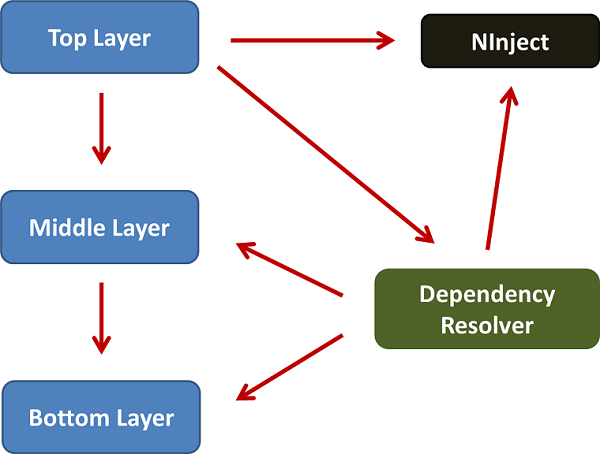


Рисунок 3.13 – Использование Ninject в приложении

### **3.11** Разработка функциональной схемы

Функциональная схема состоит из нескольких блоков, таких как:

* Плата расширения;
* Драйвер управления устройствами ввода-вывода.

Рассмотрим более подробно каждый из блоков функциональной схемы.

**3.11.1** Плата расширения

Данная плата предназначена для обеспечения расширяемости аппаратной части нашего дипломного проекта. Для обеспечения расширяемости было предпринято решение использовать сдвиговый регистр. Сдвиговый регистр позволяет нам перенести множество необходимых выходов на один выход драйвера, для этого необходимо прогрузить в сдвиговый регистр необходимые значения последовательно. Одной из особенностей сдвигового регистра является то, что мы можем собирать несколько сдвиговых регистров в один сдвиговый регистр, что позволяет нам расширять систему до тех пор, пока время заполнения сдвиговых регистров не будет превышать порога в 5 миллисекунд.

Второй задачей платы расширения является обеспечение питанием всех устройств ввода-вывода, которые будут подключаться к данной плате.

**3.11.2** Драйвер управления устройствами ввода-вывода

В нашем дипломном проекте в качестве драйвера управления устройствами ввода-вывода был выбран микроконтроллер Arduino.

Основными задачами драйвера управления устройствами ввода-вывода являются хранение информации для воспроизведения на устройствах ввода-вывода, диспетчеризация устройств ввода-вывода, синхронизация с компьютером для обмена информации. В качестве драйвера в нашем курсовом проекте использовалась Arduino Leonardo. Данная плата управляется благодаря микроконтроллеру ATmega32u4. Данная плата имеет 32 килобайта памяти, для хранения прошивки, так как для реализации алгоритмов преобразование файлов в нужный формат необходимо написать большое количество кода, который не помещается в 32 килобайта, мы реализовывали данные алгоритмы на компьютере.

**3.11.2.1** Хранение информации на драйвере

Для хранения информации на драйвере требуется иметь большой объем энергонезависимой памяти. Так как в качестве драйвера был выбран микроконтроллер Arduino, то в качестве энергонезависимой памяти мы имеем энергонезависимую память EEPROM, в зависимости от процессора, который установлен на плате Arduino мы имеем различный объем EEPROM-памяти, от 512 байт до 4 килобайт, но данной памяти недостаточно для хранения большого количество данных, которые нам необходимы. Для решения проблемы хранения большого количества данных нами было принято решение использовать внешние карты памяти, из-за их доступности и легкости в использовании.

Secure Digital-карта – это долговременный формат хранения информации для использования в портативных устройствах. Наиболее распространенным типом карт является карта формата microSD, по это причине в курсовом проекте используется карта формата microSD.

Карты могут поддерживать режимы передачи, но в данном курсовом проекте был выбран режим шины SPI, как наиболее удобный из всех представленных вариантов передачи информации. Данный тип шины поддерживает только 3.3-вольтный интерфейс. SPI – это последовательный периферийный интерфейс, с физической точки зрения SPI представляет собой синхронную четырехпроводную шину. Данный тип шип представляет собой соединение двух синхронных сдвиговых регистров, которые являются центральным элементом любого SPI устройства. Для соединения используется конфигурация, ведущий/ведомый. Только ведущий может генерировать импульсы синхронизации. В общем случае выход ведущего соединяется со входом ведомого, и наоборот, выход ведомого соединяется со входом ведущего. При подаче импульсов синхронизации на выход SCK, данные выталкиваются ведущим с выхода MOSI, и захватываются ведомым по входу MISO. Таким образом если подать количество импульсов синхронизации, соответствующее разрядности сдвигового регистра, то данные в регистрах обменяются местами. Отсюда следует что SPI всегда работает в полнодуплексном режиме. Часто бывает, что данные полученные от устройства при записи в него данных являются мусором, в таком случае их просто игнорируют, но мы их получим вне зависимости от нашего желания. Как видно на рисунке 3.14, в шине SPI используется общий сигнал выбора, ведомого для всех ведомых. Выход каждого из ведомых соединяется со входом следующего. Выход последнего ведомого соединяется со входом ведущего, таким образом образуется замкнутая цепь. При таком подключении можно считать, что последовательно соединённые устройства образуют один большой сдвиговый регистр. Соответственно, данные можно записать во все устройства за один проход, предварительно собрав нужный пакет, объединяющий данные для каждого из устройств в порядке, соответствующем физическому порядку соединения. SPI не поддерживает какого-либо механизма автоматического обнаружения устройств. К тому же, в большинстве случаев, SPI устройства не предусматривают горячее подключение/отключение.

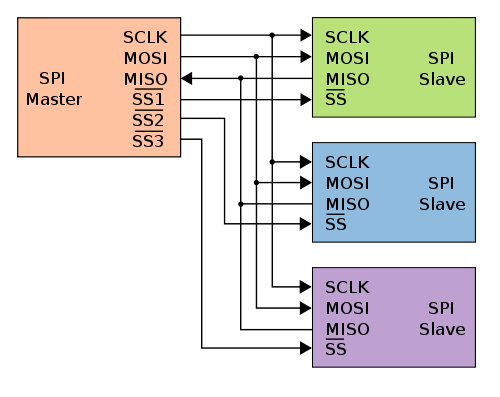


Рисунок 3.14 – Принцип работы шины SPI

Для работы с картой памяти посредством платформы Arduino в первую очередь данную карту надо отформатировать, поддерживаемые форматы библиотеками Arduino – FAT32 и FAT16. После подготовки карты памяти нам необходимо подключить карту память к драйверу, распиновка карты памяти изображена на рисунке 3.15, после этого подключив библиотеку для работы с картами памяти в программе для написания кода в Arduino, мы получим возможность работать с картой памяти. Среди возможностей библиотеки для работы с картами памяти существуют запись на карту памяти, чтение с карты памяти, проверка целостности карты памяти, создание папок и работа с файлами на карте памяти.

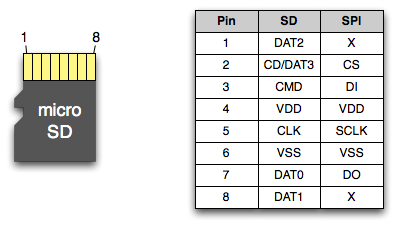


Рисунок 3.15 – Распиновка карты microSD

**3.11.2.2** Диспетчеризация устройств ввода-вывода

Для верного воспроизведения композиции нам необходимо правильно распределять воспроизводимые ноты на устройства ввода-вывода. Основные проблемы, с которыми мы столкнулись при диспетчеризации это необходимость объединять нужные ноты в случае, если количество воспроизводимых нот больше, чем количество устройств ввода-вывода. Проблемой также был выбор ноты, которую надо воспроизводить на нескольких устройствах ввода-вывода для увеличения громкости воспроизводимой мелодии.

Чтобы воспроизводить нужные нам ноты в нужном порядке, нам необходимо использовать механизмы, позволяющие нам контролировать текущее время и генерировать события с произвольным промежутком времени. Для этих целей мы используем библиотеку Arduino Timer1.

Данная библиотека представляет собой набор функций для настройки аппаратного 16-битного таймера Timer1 в ATmega32u4. В микроконтроллере доступно 3 аппаратных таймера, которые могут быть настроены различными способами для получения различных функциональных возможностей. Начало разработки данной библиотеки было вызвано необходимостью быстро и легко установить период или частоту ШИМ сигнала, но позже она разрослась, включив в себя обработку прерываний по переполнению таймера и другие функции. Она может быть легко расширена или портирована для работы с другими таймерами.

Точность таймера зависит от тактовой частоты процессора. Тактовая частота таймера Timer1 определяется установкой предварительного делителя частоты. Этот делитель может быть установлен в значения 1, 8, 64, 256 или 1024. На рисунке 3.16 показана работа таймера при тактовой частоте 16 мегагерц.



Рисунок 3.16 – Работа таймера при частоте 16 мегагерц

**3.11.2.3** Синхронизация драйвера с компьютером

Для синхронизации драйвера с компьютером используется библиотека Firmata. Данная библиотека уже была описана в данном разделе, однако стоит добавить, что для полноценного функционирования, данная библиотека должна быть установлена на платформе Arduino. Firmata поддерживает различные виды подключений к компьютеру, однако в дипломном проекте использовалось serial-подключение посредством usb-шнура, так как это подключение является наиболее стабильным.

# **4** рАЗРАБОТКА ПРОГРАММНЫх МОДУЛЕЙ

Во время разработки приложений разработчику помимо использования стандартных библиотек и фреймворком приходиться реализовывать некоторое количество нестандартных алгоритмов. В данном разделе будут рассмотрены нестандартные алгоритмы, реализованные в данном дипломном проекте.

### **4.1** Алгоритм преобразования входных музыкальных файла

В программной части дипломного проекта реализована обработка MIDI-, MP3- и WAV-файлов. Так как суть классов обработчиков в независимости от типа файла одна и та же, преобразовать входной файл в массив данных, был реализован интерфейс IConvertService, который имел единственный метод GetNotes, возвращающий массив структур NoteEntity.

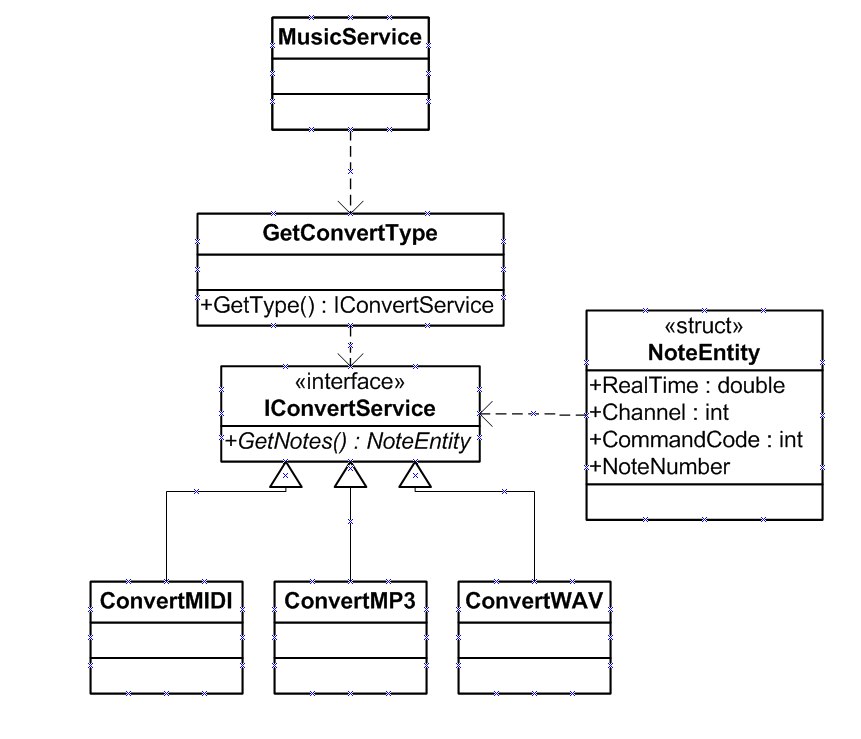


Рисунок 4.1 – Диаграмма взаимодействия классов

Структура NoteEntity имеет ряд полей, который необходимы нам для воспроизведения композиции на устройствах ввода-вывода, а именно:

* RealTime – время воспроизведения композиции;
* NoteNumber – номер воспроизводимой композиции;
* Channel – номер устройства для воспроизведения;
* CommandCode – код события.

Во время загрузки файла в приложение, при его обработке, учитывается его тип файла, и при помощи фабрики GetConvertType, которая реализована в виде статического класса, со статическим методом GetType, принимающий тип файла в виде строки, мы получаем класс, реализующий интерфейс IConvertService, который в свою очередь преобразовывает входной файл. Рассмотрим одну из реализаций интерфейса IConvertService класс ConvertMIDI.

Для преобразования MIDI-файла в последовательность событий необходимо совершить несколько действий, а именно проанализировать MIDI-файл и вычленить из него события, из полученного массива событий оставить только необходимые для воспроизведения на устройствах ввода-вывода события, организовать события в нужном порядке и с необходимой частотой. Рассмотрим данные шаги алгоритма преобразования подробнее.

**4.1.1** Структура и работы класса ConvertMIDI

Для анализа и дальнейшего разбора MIDI-файла была использована библиотека NAudio, архитектура данной библиотеки показана на рисунке 4.2. В данном случае был использован класс MidiFile, конструктор которого принимает два параметра, первый это путь к файлу, который необходимо преобразовать. Вторым параметром является булевое значение, которым мы указываем необходимость проверки структуры MIDI-файла, в случае несоблюдения структуры MIDI-файла будет сгенерирована ошибка FormatException. Объект данного класса позволяет нам в дальнейшем работать с MIDI-файлом.

Чтобы верно преобразовать файл нам необходимо получить массив типа NoteEntity, для этого нем необходимо получить два массива, первый отвечает за включения и отключения нот, второй отвечает за изменения темпа композиции, а далее совместить два этих массива.

Для того чтобы получить все необходимые нам события включения и выключения нот, мы реализовали private-метод BuildAbsoluteNotes, при помощи данного метода мы обходим все события файла, которые хранятся в поле CollectionEvent, объекта класса MidiFile. Нам необходимы только события с кодами 144 и 128, это события включения и отключения ноты. Так как в коллекции CollectionEvent события перечислены не по времени воспроизведения, а по каналам, в которых данное событие срабатывает, то появляется необходимость отсортировать события по полю AbsoluteTime. В данном поле хранится время выполнения события в тиках. Для того чтобы преобразовать тики в секунды, нам необходимо построить массив изменения темпа. Для построения массива изменения темпа мы реализовали метод BuildTempoList.

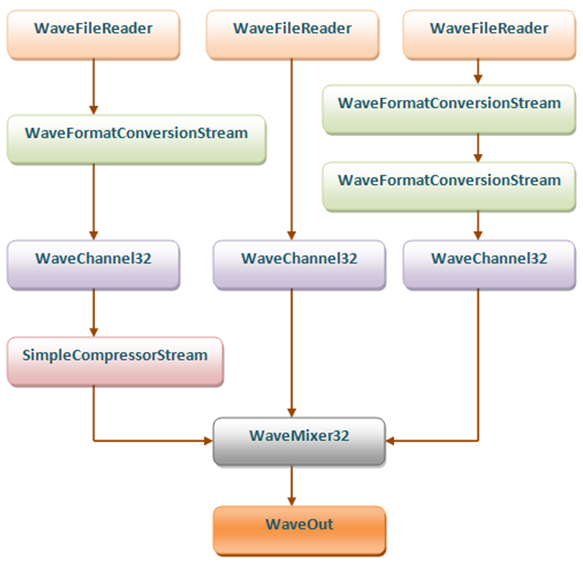


Рисунок 4.2 – Архитектура библиотеки NAudio

В методе BuildTempoList строится массив изменения темпа, для этого мы должны учитывать события изменения темпа в MIDI-файле. Чтобы получить все события изменения темпа мелодии, нам необходимо из всех событий MIDI-файла выбрать события с кодом 81. При изменении темпа у нас изменяется значение времени, которое означает один тик, стало быть нам требуется хранить это изменение, для этого используется структура NoteEntity.

После того как мы получили массив нот и массив изменения темпа, мы можем получить массив, в котором будет указано событие включения или выключения ноты на определенном канале и время в миллисекундах, когда данное событие должно произойти. Ниже демонстрируется код, который преобразует тики в миллисекунды.

int counter = 0;

double currentRealTime = 0.0;

double lastRealTime = 0.0;

List<NoteEntity> output = new List<NoteEntity>(Count);

foreach (var absoluteNote in absoluteNotes)

{

counter++;

currentRealTime = GetRealtime(AbsoluteTime);

output.Add(new NoteEntity()

{

RealTime = currentRealTime,

NoteNumber = absoluteNote.NoteNumber,

Channel = absoluteNote.Channel,

CommandCode = (int)absoluteNote.CommandCode

});

lastRealTime = currentRealTime;

}

Полученными данными можно успешно воспользоваться для воспроизведения музыкальных композиций на устройствах ввода-вывода.

### **4.2** Реализация Plug and Play подключения

Для реализации Plug and Play в нашем приложении использовался класс DeviceWatcher, конструктор которого принимает строковый фильтр, указывающий какие именно порты необходимо прослушивать. В нашей реализации, DeviceWatcher прослушивает только COM-порты, так как платформа Arduino подключается к компьютеру при помощи виртуального COM-порта. Виртуальный COM-порт позволяет имитировать присутствие последнего в компьютере.Кроме того, мы подписываемся на события Added и Removed, что позволяет нам получать информацию о подключенных устройствах и устройствах, которые мы отключили.

Для того чтобы отображать изменение количества подключенных устройств на графическом интерфейсе, был создан класс DeviceInformations, который хранит в себе информацию об устройстве и его шаблоне на графическом интерфейсе. В данном классе содержаться свойства:

* DeviceInformation – хранит в себе информацию об подключенном устройстве;
* RemoteDevice – содержит набор полей для управлением драйвером;
* StackPanel – хранит в себе информацию о панели, где будет отображаться информация об состоянии устройстве;
* Serial – хранит в себе информацию о потоке, при помощи которого будет производиться обмен информации между драйвером и компьютером.

В том момент, когда мы подключаем устройство к системе, срабатывает событие Add, в данном событии происходит конфигурирование устройства, в рамках которого мы добавляем панель для отображения информации об устройстве, создаем поток при помощи которого будем производить обмен сообщениями между драйвером и компьютером. Ниже представлен код события Add.

await this.dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

StackPanel stackPanel = AddingStackPanel(device.Id);

DevicePanel.Children.Add(stackPanel);

IStream usbSerial = new UsbSerial(device);

RemoteDevice arduino = new RemoteDevice(usbSerial);

usbSerial.begin(57600, SerialConfig.SERIAL\_8N1);

DeviceInformations deviceInformations = new DeviceInformations(device, usbSerial, arduino, stackPanel);

devices.Add(deviceInformations);

});

Во время отключения устройства, происходит высвобождения занятых неуправляемых ресурсов в виде потока, а также удаление панели информации об устройстве из графического интерфейса. Ниже представлен код события Remove.

await this.dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>

{

var removeDevice = devices.FirstOrDefault(x => x.DeviceInformation.Id == device.Id); DevicePanel.Children.Remove(removeDevice.StackPanel);

devices.Remove(removeDevice);

});

На рисунке 4.3 показана процедура обмена сообщениями между платформой Arduino и устройством с операционной системой Windows 10. Стоит отметить, так как события Add и Remove срабатывают в потоке, отличным от потока прорисовки графического интерфейса, то нам приходиться запускать операцию обновления графического интерфейса в новом потоке, с заданием стандартного приоритета выполнения.

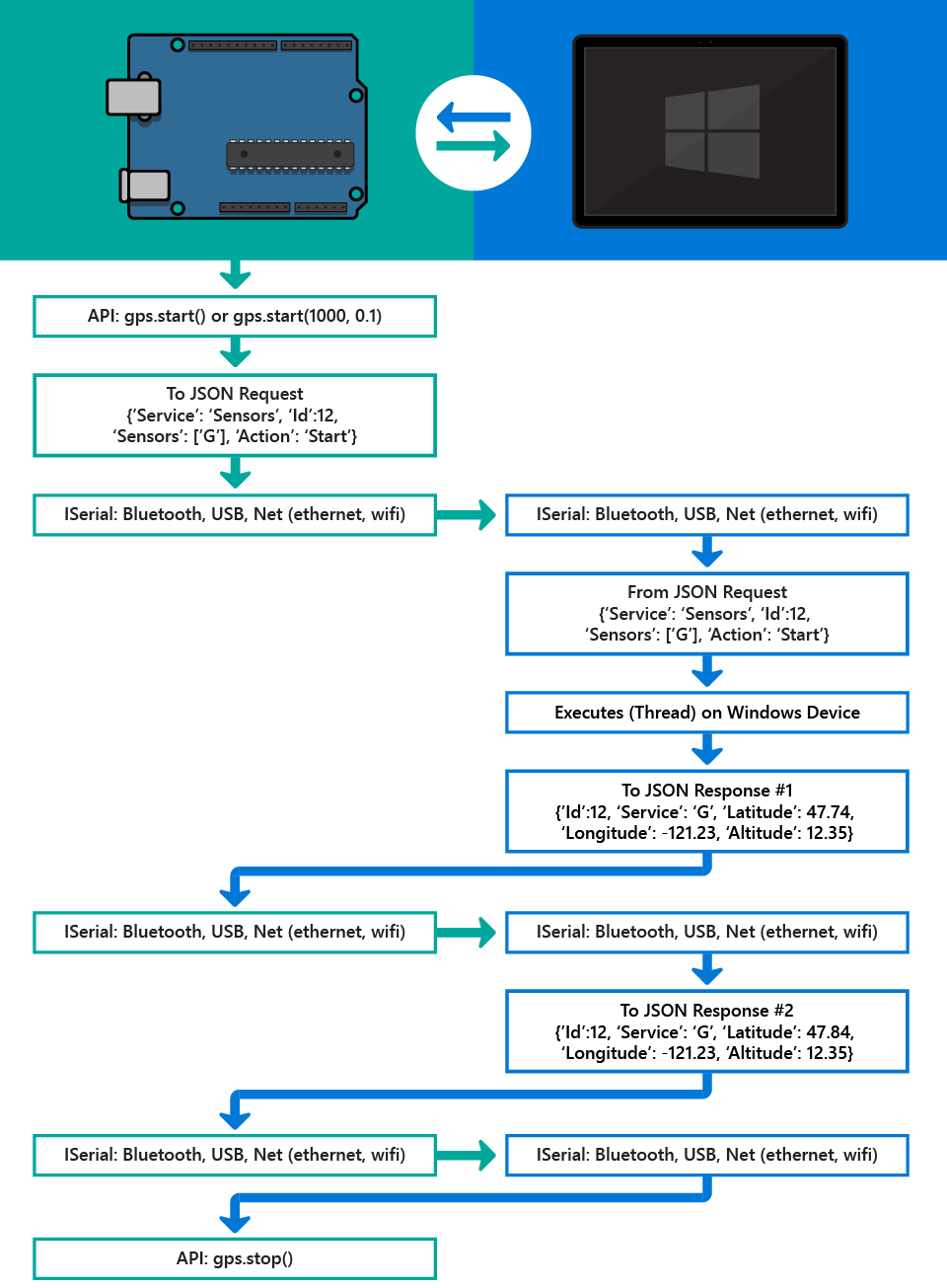


Рисунок 4.3 – Принцип обмена сообщений между устройствами

### **4.3** Реализация паттерна unit of work

Паттерн unit of work позволяет нам инкапсулировать логику работы с источниками данных. Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.

Для реализации данного паттерна в первую очередь был создан интерфейс IUnitOfWork, который в свою очередь реализует интерфейс IDisposable. Основное применение интерфейса IDisposable состоит в освобождении неуправляемых ресурсов. Сборщик мусора автоматически освобождает память, выделенную для управляемого объекта, если этот объект больше не используется. Тем не менее не может предсказать, когда будет выполнена сборка мусора. Кроме того, сборщик мусора не имеет сведений о неуправляемые ресурсы, таких как дескрипторы окон, открытие файлов и потоков или соединения с базой данных как в нашем случае. Рассмотрим более подробно методы интерфейса IUnitOfWork, которые должен реализовать класс выполняющий данный интерфейс:

* Save – сохраняет изменения произошедшие в контексте базы данных;
* Dispose – освобождает неуправляемые ресурсы, в данном случае контекст базы данных.

После определения интерфейса необходимо создать класс, который будет данный интерфейс реализовывать. В нашем проекте мы создали класс UnitOfWork, который производил все операции с базой данных. Данный класс дополнительно содержит ряд свойств, который необходимо перечислить для полного понимания данного паттерна.

Поле Context содержит контекст базы данных, именно через него выполняют все запросы в базу данных. Так как контекст базы данных является неуправляемым ресурсом, нам необходимо переопределить метод Dispose интерфейса IDisposable, для того чтобы при удалении класса, либо использовании данного класса в директиве using, мы могли бы освобождать ресурсы.

Поле IRepository является интерфейсом, который реализуется для каждой таблицы в базе данных, позволяя совершать нам ряд операций с конкретными таблицами в базе данных, а именно:

* Create – при помощи контекста, позволяет добавлять новую запись в таблицу базы данных;
* Read – при помощи контекста, позволяет возвращать существующую запись из таблицы базы данных;
* Update – при помощи контекста, позволяет обновлять существующую запись в таблице базы данных;
* Delete – при помощи контекста, позволяет удалять существующую запись в таблице базы данных;

В дальнейшем мы можем используя класс UnitOfWork через директиву using, совершать множество операций над таблицами в базах данных и при этом только единожды переносить информацию из контекста базы данных в базу данных, при помощи метода Save(). На рисунке 4.4 показан принцип работы паттерна unit of work.

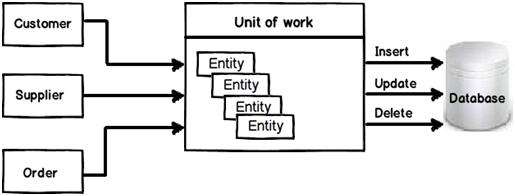


Рисунок 4.4 – Принцип работы паттерна unit of work

### **4.4** Реализация принципа инверсии управления

Для реализации данного принципа в приложение была добавлена библиотека Ninject Portable, данная библиотека предназначена для десктопных приложений. Архитектура библиотеки Ninject Portable показана на рисунке 4.5.

Для того, чтобы воспользоваться возможностями библиотеки Ninject Portable мы добавили в наше решение еще один проект, который назвали ResolveModule, добавив связи ко всем проектам. В данном проекте мы реализовали единственный класс ResolveConfig, в котором содержится один метод Configure, который принимает в качестве входного параметра ядро библиотеки Ninject. В данном методе происходит настройка ядра приложения, и сопоставление интерфейсов их реализациям. Для настройки ядра используются метод Bind, который создает связь между интерфейсом и реализацией данного интерфейса. Кроме метода Bind, существуют также методы InSingletonScope и InTransientScope, рассмотрим данные методы более подробно.

InSingletonScope данный метод указывает, что создается только один объект реализации интерфейса, и он используется при всех вызовах данного объекта. Данный метод является реализацией паттерна Singleton. Паттерн Singleton позволяет создавать только один объект определенного класса.

InTransientScope данный метод указывает, что необходимо создавать объект реализации каждый раз когда мы к нему обращаемся, повторное использование объекта не предусматривается.

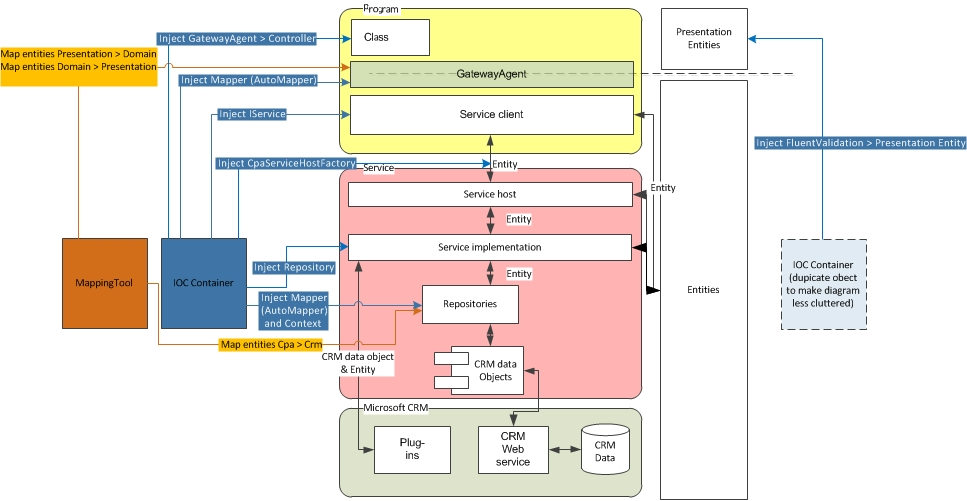


Рисунок 4.5 – Архитектура библиотеки Ninject Portable

Ниже продемонстирован код настройки ядра библиотеки Ninject Portable.

kernel.Bind<IUnitOfWork>().To<UnitOfWork>().InSingletonScope(); kernel.Bind<DbContext>().To<Context>().InSingletonScope(); kernel.Bind<DbContext>().To<Context>().InTransientScope(); kernel.Bind<IMusicRepository>().To<MusicRepository>(); kernel.Bind<IMusicService>().To<MusicService>();

Метод ResolveConfig класса ResolveModule вызывается на верхнем уровне нашей иерархии, в приложении, реализовывающем графический интерфейс.

# **5** Программа и методика испытаний

### **5.1** Выбор средств и методологии тестирования

Тестирование программного обеспечения на текущий момент является неотъемлемой частью его разработки. Современные методы тестирования позволяют не только обнаруживать ошибки, но также и выявлять причины их появления, что ускоряет процесс их устранения.

Тестирование программного обеспечения – это процедура, которая поз­воляет подтвердить или опровергнуть работоспособность кода и коррект­ность его работы. При тестировании классу или методу передаются входные данные и запрашивается выполнение некой команды, после чего произво­дится проверка полученных результатов на соответствие эталону, если результат соответствует ожидаемому – тест считается пройденным. Эта про­цедура может быть автоматизирована, в этом случае проверка работоспособ­ности и правильности работы приложения осуществляется гораздо быстрее, полноценнее и чаще (в сравнении с ручным тестированием).

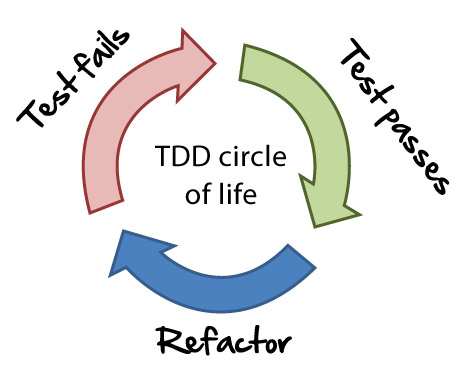


Рисунок 5.1 – Методология тестирования программного обеспечения

Методика разработки через тестирование заключается в организации автоматического тестирования разрабатываемых приложений путем написа­ния модульных, интеграционных и функциональных тестов, определяющих требования к коду непосредственно перед написанием этого самого кода. Сначала пишется тест, который проверяет корректность работы еще ненапи­санного программного кода. Этот тест, разумеется, не проходит. После этого разработчик пишет код, который выполняет действия, требуемые для про­хождения теста. После того, как тест успешно пройден, по необходимости осуществляется рефакторинг (доработка и переработка) написанного кода, причём рефакторинг осуществляется под контролем прохождения тестов. На рисунке 5.1 изображена данная методология.

Архитектура программных продуктов, разрабатываемых таким обра­зом, обычно лучше (в приложениях, которые пригодны для автоматического тестирования, обычно очень хорошо распределяется ответственность между компонентами, а выполняемые сложные процедуры декомпозированы на множество простых). Стабильность работы приложения, разработанного через тестирование, выше за счёт того, что все основные функциональные возможности программы покрыты тестами и их работоспособность посто­янно проверяется. Сопровождаемость проектов, где тестируется всё или практически всё, очень высока – разработчики могут не бояться вносить изменения в код, если что-то пойдёт не так, то об этом сообщат результаты автоматического тестирования.

На рисунке 5.2 показаны фазы тестирования программного обеспечения, так как в данном дипломном проекте отсутствуют заказчики, фаза приемочного тестирования совпадет с фазой системного тестирования. Как видим для полноты тестирования необходимо пройти 3 фазы тестирования, рассмотрим каждую из фаз подробнее.

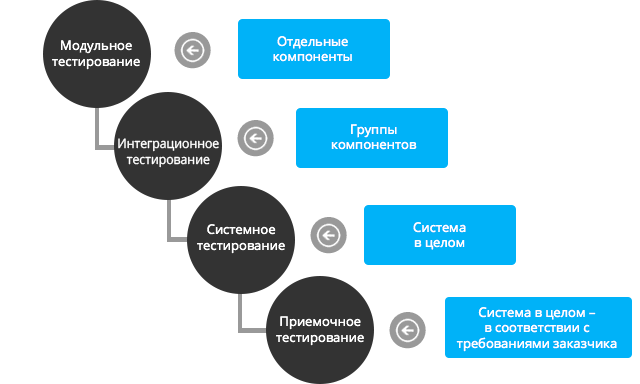


Рисунок 5.2 – Фазы тестирования программного обеспечения

### **5.2** Модульное тестирование

Компания Microsoft разработала технологию Live Unit Testing, позволяющую автоматически выполнять все модульные тесты в фоновом режиме и отображать результаты их выполнения, это намного упрощает жизнь разработчику, позволяя ему не отвлекаться на выполнение тестов, а просто наблюдать за изменением в выполнениях тестов во время написания кода. Во время изменения кода Visual Studio предоставляет отчет о том, как изменился код и как это повлияло на выполнение модульных тестов, а также указывает насколько сильно наши тесты покрывают новый код, благодаря этому разработчик не будет забывать писать новые тесты. Результат работы данной технологии изображен на рисунке 5.3. На рисунке 5.4 изображен подробный отчет по ошибке. Для корректной работы данной технологии ее необходимо настроить, из возможных настроек мы можем указать как часто будет проводиться тестирования, интервал времени, когда будет ожидаться окончания теста, количество процессов, создаваемых для тестирования, а также информацию которая будет выводиться в информационное окно.

### Изображение

Рисунок 5.3 – Результаты работы Live Unit Testing

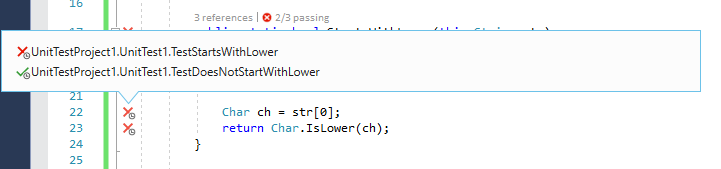


Рисунок 5.4 – Результаты работы Live Unit Testing

Данная технология способна поддерживает такие тестовые платформы, как xUnit.net, NUnit, MSTest. В данном дипломном проекте мы использовали тестовую платформу MsTests, так как на данный момент это является единственным решением компании Microsoft для тестирования UWP приложений. Но мы не можем обойтись только модульным тестированием, так как модульное тестирование затрагивает только определенные методы и функции, как изолированные сущности, но нам также необходимо проверить результаты их взаимодействия, для этого было использовано интеграционное тестирование.

### **5.3** Интеграционное тестирование

Интеграционное тестирование – одна из фаз тестирования программного обеспечения, при которой отдельные программные модули объединяются и тестируются в группе. Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию. Интеграционное тестирование в качестве входных данных использует модули, над которыми было проведено модульное тестирование, группирует их в более крупные множества, выполняет тесты, определённые в плане тестирования для этих множеств, и представляет их в качестве выходных данных и входных для последующего системного тестирования. Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным и требованиям надежности.

### **5.4** Системное тестирование

Системное тестирование – одна из фаз тестирования программного обеспечения, предназначенное для тестирование готового программного обеспечения. На данном этапе тестирования, кроме проверки функциональных возможностей, также проверяются возможности системы, а именно ее надежность, устойчивость, безопасность. Так как системное тестирования является одним из самых сложных методов тестирования, для его выполнения привлекают группу тестировщиков, так как при выполнение дипломного проекта мы не имели группу тестировщиков, системное тестирование выполнялось методом постоянной работой с приложением, а также мониторингом системы средствами Visual Studio. На рисунке 5.5 изображены средства диагностики Visual Studio 2015, по которым мы можем понять, как часто происходит сборка мусора, как много памяти требуется для процессора, а также загруженность самого процессора в необходимые для нас моменты времени.

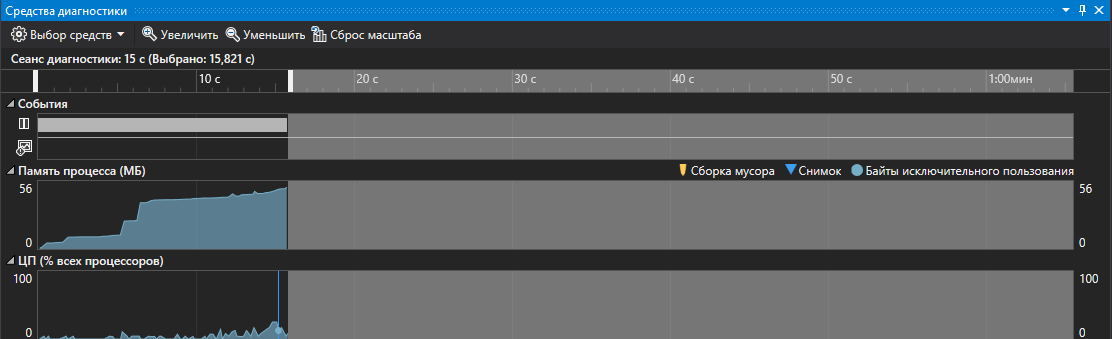


Рисунок 5.5 – Средства диагностики Visual Studio 2015

### **5.5** Обработка исключительных ситуаций

Существуют ситуации, когда после множества пройденных тестов, приложение может выдать пользователю ошибку к примеру, при отключении от базы данных, при отсутствии фрагментов фреймворка и др. В такой ситуации должна срабатывать обработка исключительных ситуаций. В нашем программном продукте обработка исключительных ситуаций реализована снизу–вверх, то есть, когда возникает ошибка на нижнем уровне, она обрабатывается и передается более высокому уровню и так до тех пор, пока не дойдет до уровня прорисовки пользовательского интерфейса, где уже будет создан элемент пользовательского интерфейса, который будет указывать на то, что во время работы приложения произошла ошибка. На рисунке 5.6 показано окно обработки исключительных ситуаций.

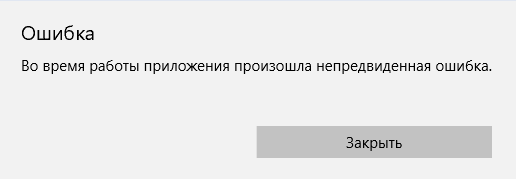


Рисунок 5.6 – Окно обработки исключительных ситуаций

# **6** Руководство пользователя

## **6.1** Установка программы на персональный компьютер

# ЗАКЛЮЧЕНИЕ

В рамках преддипломной практики была изучена предметная область, рассмотрены существующие аналоги, разработана плата для увеличения количества подключенных устройств, разработана прошивка для драйвера, а также методы взаимодействия между драйвером и персональным компьютером.

Благодаря модульной структуре платы расширения возможно дальнейший рост функциональности, главным направлением для которого является добавление поддержки новых устройств ввода-вывода.

# Список использованных источников

1. Wikipedia [Электронный ресурс]. – Режим доступа: <https://wikipedia.org/>.
2. Энциклопедический словарь музыканта / Сост. В. В. Медушевский, О. О. Очаковская. – М.: Педагогика, 1985. – 352 с.

# Музыка на дисководе гибких дисков: теория и примеры / Geektimes [Электронный ресурс]. – Режим доступа : https://geektimes.ru/post/257418/.

1. Подробнее о формате MIDI [Электронный ресурс]. – Режим доступа : <http://www.audacity.ru/p10aa1.html>/.
2. SammyIAm/Moppy [Электронный ресурс]. – Режим доступа : <https://github.com/SammyIAm/Moppy>/.
3. MIDI плеер на восьми Floppy [Электронный ресурс]. – Режим доступа : <https://habrahabr.ru/post/253189/>.
4. Албахари Б. C# 6.0. Справочник. Полное описание языка / Б. Албахари, Дж. Албахари. – М.: Вильямс, 2016. – 1054 c.
5. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж. Рихтер. – СПб.: Питер, 2016. – 896 с.
6. Intro to the Universal Windows Platform [Электронный ресурс]. –Режим доступа : https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide/.
7. Бен-Ган И. Microsoft SQL Server 2012. Основы T-SQL / И. Бен-Ган. – М.: Эксмо, 2015. – 400 с.

# ПРИЛОЖЕНИЕ А

(*справочное*)

Вводный плакат

# ПРИложение Б

(*справочное*)

Структурная схема