МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ

ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»



Направление подготовки   
09.03.01 Информатика и вычислительная техника

Направленность (профиль)

«Технологии разработки программного обеспечения»

**Выпускная квалификационная работа**

Создание MIDI контроллера на основе микроконтроллера Arduino

|  |
| --- |
| Обучающегося 4 курса  очной формы обучения  Егорова Сергея Андреевича |
|  |
| Руководитель выпускной квалификационной работы:  кандидат физико-математических наук,  доцент кафедры информационных технологий и электронного обучения  Власов Дмитрий Викторович |
|  |

Санкт-Петербург

2024

Оглавление

[ВВЕДЕНИЕ 3](#_Toc154185792)

[Глава 1.](#_Toc154185793)

**ВВЕДЕНИЕ**

Актуальность дипломной работы связана с появлением и развитием программных цифровых рабочих аудио станций (Digital Audio Workstation, далее DAW). Они отрыли возможность для создания собственных низкобюджетных студий на основе персонального компьютера для людей, работающих со звуком как на любительском, так и на профессиональном уровне. При взаимодействии с DAW большую роль играют специализированные устройства ввода, позволяющие эффективно контролировать различные параметры, существующие в рамках работы с аудио материалом. На фоне изначально высокой стоимости большинства таких устройств и ввиду отсутствия отечественных аналогов в условиях затрудненного импорта, возникает необходимость исследования и разработки максимально доступного решения с открытым исходным кодом, которое позволит всем заинтересованным людям с достаточным уровнем навыков дешево и быстро воспроизводить и модифицировать такое устройство. Оптимальным выбором аппаратной платформы для этого устройства являются микроконтроллеры типа Arduino, ввиду возможности прошивать такие платы от разных производителей без необходимости изменения исходного кода.

Объектом исследования дипломной работы является разработка устройств на аппаратной платформе Arduino. Предметом является разработка MIDI котроллера на основе микроконтроллера типа Arduino и его интеграция в программные цифровые рабочие аудио станции.

Целью дипломной работы является создание совместимого с современными DAW MIDI контроллера, предназначенного для регулирования громкости аудио каналов, с возможностью переключения между группами каналов и обратной связью с ПК, позволяющей синхронизировать состояние устройства с цифровой рабочей аудио станцией.

Задачами дипломной работы в связи с указанной целью являются:

1. Исследовать рынок электронных компонентов и найти наиболее доступные предложения без существенной потери качества;
2. Разработать механизм моторизации скользящего потенциометра, необходимого для корректного контроля уровней громкости внутри DAW;
3. Разработать электрическую схему подключения компонентов устройства;
4. Разработать макет печатной платы на основе полученной электрической схемы.
5. Разработать чертеж и модель для трехмерной печати корпуса устройства;
6. Разработать ПО под выбранную аппаратную платформу для управления устройством.
7. Разработать ПО для интеграции устройства в DAW.

В результате написания дипломной работы планируется получить рабочий прототип такого устройства, пригодный и доступный к воспроизведению и модификации.

В рамках исследования планируется использовать следующие методы:

1. Анализ научной литературы;
2. Анализ технической документации;
3. Анализ исходного кода похожих решений;
4. Прототипирование;

Практическая значимость дипломной работы состоит в нахождении способа создания дешевого решения для производства такого устройства в условиях персонального проекта, что может сделать вхождение в сферу работы с аудио доступнее для заинтересованных в этом людей.

В введении указана актуальность исследовательской работы, ее объект, предмет цель и задачи. А также гипотеза методы исследования и практическая значимость. В первой главе рассматриваются теоретические аспекты аппаратной и программной составляющей устройства. Во второй главе рассматривается практическая реализация сборки устройства и написания для него ПО, а также написание ПО для интеграции в программные цифровые рабочие аудио станции. В заключении подводятся итоги исследования. Проводится оценка стоимости полученного устройства и возможностей его дальнейшего улучшения. Формулируются окончательные выводы по теме дипломной работы. ­

**ОСНОВНАЯ ЧАСТЬ**

Глава 1. Теоретическая часть.

* 1. Выбор электронных компонентов.

В первую очередь был произведен выбор модели микроконтроллера. Главной его характеристикой в рамках данной работы является поддержка протокола коммуникации USB. Она в купе с использованием специализированных библиотек позволяет подключать микроконтроллер сразу как устройство класса MIDI, автоматически распознаваемое API большинства современных операционных систем. Однако для удобства в ходе диплома подключение будет рассматриваться на примере операционной системы Windows как наиболее популярной для разнообразных мультимедийных задач. Эта характеристика существенно сокращает выбор в рамках модельной линейки микроконтроллеров Arduino. Такую возможность поддерживают только платы на основе микропроцессоров ATmega16U2 и ATmega32U4. Однако использование моделей с ATmega16U2 требует дополнительной настройки, без какого-либо преимущества с точки зрения конечного результата. По этой причине выбор микроконтроллеров семейства Arduino для создания MIDI устройства сводится к моделям Leonardo, Micro и Micro Pro. Так как модель Leonardo предназначена для беспаечных прототипов, а модель Micro Pro отличается от Micro размером за счет отсутствия малозначимой кнопки сброса на поверхности платы, был выбран микроконтроллер Arduino Micro Pro (Таблица 1).

Таблица 1 – характеристики микроконтроллера Arduino Micro Pro

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Микропроцессор | ATmega32u4 |
| Предельное напряжение питания, В | 5-20 |
| Рекомендуемое напряжение питания, В | 7-12 |
| Цифровых вводов/выводов | 18 |
| ШИМ выводов | 5 |
| Аналоговых вводов/выводов | 9 |
| Flash память, КБ | 32 |
| Тактовая частота, МГц | 16 |

Ключевым компонентом для регулирования громкости в аналоговых и цифровых микшерных пультах является потенциометр. За редким исключением для этой цели всегда используются скользящие потенциометры (потенциометры движкового типа). В сущности, любой потенциометр является резистором переменного сопротивления, и обладает следующими важными для рассматриваемой задачи характеристиками: ход движка, номинал, характеристика и рабочее усилие на движке. Минимальный необходимый ход движка для комфортного управления громкостью на профессиональных пультах зачастую начинается от 60мм. Номиналом называется диапазон возможных значений сопротивления. Наиболее распространенным является 10КОм ±20%, что полностью подходит для цифровой реализации управления громкостью, ведь это сопротивление не влияет на мощность сигнала напрямую, как это происходит в аналоговых микшерных пультах. По той же причине из двух возможных характеристик – линейной и логарифмической, гораздо удобнее выбрать первую, так как логарифмическое изменение мощности сигнала необходимо только при обработке аналогово аудио сигнала, ввиду того что в основе шкалы измерения децибел лежит десятичный логарифм. В случае цифровой обработки конвертацию в логарифмическую шкалу DAW делают сами, принимая на вход линейное значение, отражающее значение уровня громкости канала. Под рабочим усилием имеется ввиду необходимая сила для сдвигания ручки потенциометра. Она измеряется в грамм-силе и может сильно варьироваться в зависимости от модели и производителя, однако в рамках данной работы важно чтобы только она не была слишком большой, чтобы не поднимать стоимость необходимого для моторизации электродвигателя. Таким образом момент силы искомого потенциометра можно выразить как 20±5 гс. По результатам поиска самой оптимальной моделью стал потенциометр RA6021F-20-15C1-B10K-C1 от производителя Alpha (таблица 2).

Таблица 2 – характеристики потенциометра Alpha RA6021F-20-15C1-B10K-C1

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Размеры корпуса (ДШВ) , мм | 75.0x9.0x6.5 |
| Высота движка, мм | 15 |
| Ход движка, мм | 60 |
| Номинал, КОм | 10±20% |
| Характеристика | B (линейная) |
| Рабочее усилие на движке, гс | 20.200 |

Вторым по значимости компонентом моторизованного потенциометра является электромотор. По результатам исследования конструкций существующих решений, был сделан вывод, что для данной задачи достаточно обычного коллекторного электродвигателя постоянного тока без использования редуктора. Опираясь на характеристику рабочего усилия на движке потенциометра, а также на габариты и вольт амперную характеристику, был выбран электродвигатель F280-15200 12V производителя RUICHI (Таблица 3).

Таблица 3 – характеристики электродвигателя RUICHI F280-15200 12V

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Тип двигателя | двигатель постоянного тока |
| Размеры корпуса (ДШВ) , мм | 45.0x20.0x24.2 |
| Расчетное напряжение, В | 12 |
| Расчетный ток, A | 0.41 |
| Частота вращения под нагрузкой, об/мин | 7909 |
| Момент, гс cм | 36,6 |

Все микроконтроллеры типа Arduino не поддерживают 12 Вольт, необходимых для выбранной модели электромотора, в качестве выходного напряжения. Исходя из этого возникает необходимость во внешнем питании для корректной работы моторизованных потенциометров. Для решения этой задачи были рассмотрены два варианта источников питания. Блок батарей формата АА и сетевой адаптер питания с выходом на разъем DC. Из полученных данных по результатам замеров выходного напряжения и силы тока у обоих опций, было произведено сравнение мощности источников питания с необходимой мощностью для одновременной работы 4 электродвигателей, вычисленной по следующей формуле.

Таким образом необходимая мощность равняется 19,2 Ватт. На основании этого был сделан выбор в пользу адаптера питания. Так же стоит отметить, что использования адаптера питания вместо блока батарей может существенно упростить дальнейшее обслуживание устройства и повысить его отказоустойчивость. В рамках выполнения данной дипломной работы был использован блок питания FT-1220-AC/DC-12V-2A от производителя FIBO (Таблица 4).

Таблица 4 – характеристики блока питания FIBO FT-1220-AC/DC-12V-2A

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Выходное напряжение, В | 12 |
| Сила тока, A | 2 |
| Расчетное напряжение, В | 12 |
| Мощность, Вт | 24 |
| Формат разъема DC, мм | 5,5-2,5 |

Для подачи тока на моторы, с возможностью удобно управлять скоростью и направлением вращения, необходимо использовать драйверы моторов. В основе любого драйвера лежит электронная схема под названием Н-мост (Рисунок 1). Она позволяет изменять полярность применяемого тока.

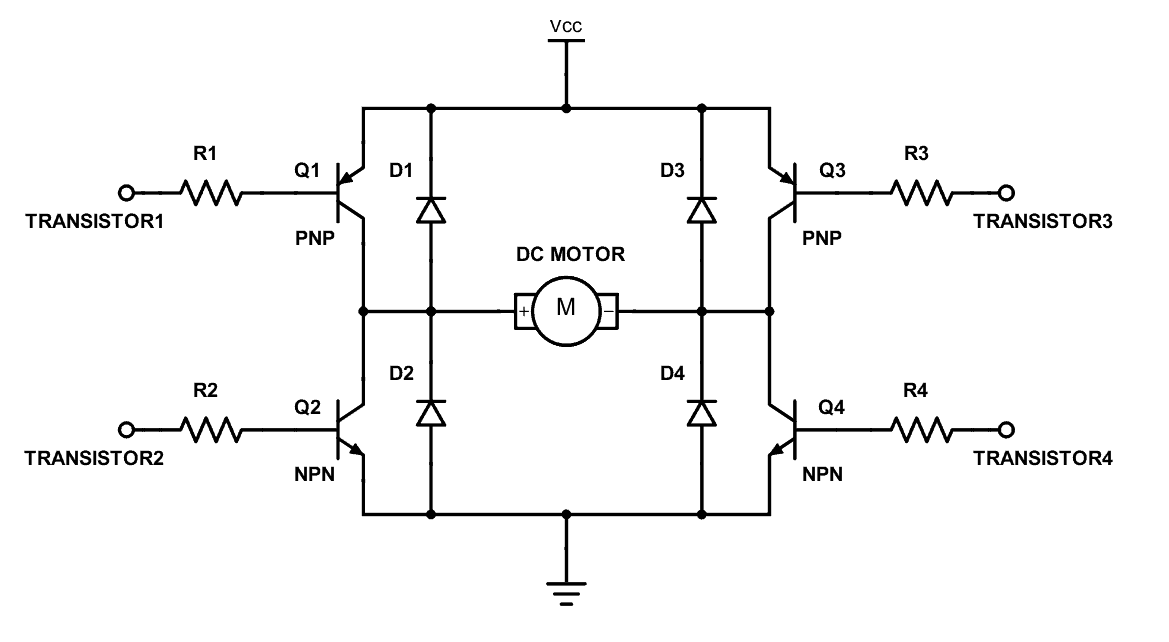


Рисунок 1. Электросхема Н-моста.

Отличие драйвера от Н-моста состоит в наличии дополнительной надстройки в виде удобного интерфейса для подключения к микроконтроллерам с целью подачи обычного цифрового и ШИМ (Широтно-Импульсно Модулированного) сигнала, механизмов защиты от перегрева и возможности подключения сразу нескольких моторов. При сборке тестового прототипа были рассмотрены два варианта подачи тока на моторы - драйвер на 4 мотора на базе чипа L293D, и два драйвера на два мотора на базе чипа TA6586 (Таблица 5). В результате замеров более современный чип TA6586 показал гораздо большую эффективность и меньшую склонность к нагреву даже при продолжительной работе. На основании этого был сделан выбор в пользу двух драйверов, сделанных на его базе.

Таблица 5 – характеристики драйвера моторов на базе чипа TA6586

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Рабочее напряжение, В | 3-14 |
| Входное напряжение сигнала, В | 2,2-6 |
| Ток привода, A | 5 |

Кроме потенциометров, конструкция устройства предполагает использование двух тактовых кнопок, необходимых для переключения активной группы регулируемых котроллером аудио каналов. Данные компоненты являются наименее требовательными из всех, в следствии чего была выбрана простая модель тактовой кнопки TC-12ET (Таблица 6) от производителя KLS.

Таблица 6 – характеристики тактовой кнопки KLS TC-12ET

|  |  |
| --- | --- |
| Наименование характеристики, единица измерения | Значение |
| Рабочее напряжение, В | 12 |
| Рабочий ток, А | 0.05 |
| Размеры (ДШВ), мм | 12x12x7 |

Таким образом был составлен список необходимых компонентов для создания MIDI контроллера предназначенного для регулирования громкости аудио каналов с возможностью динамического переключения между назначенными на потенциометры виртуальными каналами, поддерживаемый медиа API большинства современных ОС как MIDI устройство по умолчанию. Так же немаловажную роль на этапе раннего прототипирования сыграла беспаечная плата, позволяющая быстро собирать различные участки электрической цепи предполагаемого устройства для тестирования и отладки.

* 1. Планирование подключения компонентов к пинам микроконтроллера.

Для структурирования всей дальнейшей работы и понимания необходимого количества задействованных в устройстве пинов микроконтроллера были составлены следующие таблицы.

Таблица 7 – подключение потенциометров.

|  |  |
| --- | --- |
| Пин микроконтроллера | Номер потенциометра |
| A0 | 1 |
| A1 | 2 |
| A2 | 3 |
| A3 | 4 |

Таблица 8 – подключение драйверов моторов.

|  |  |
| --- | --- |
| Пин микроконтроллера | Номер драйвера - пин |
| D2 | 1 - D1 |
| D3(PWM) | 1 - D0 |
| D4 | 1 - D3 |
| D5(PWM) | 1 - D2 |
| D6(PWM) | 2 - D0 |
| D7 | 2 - D1 |
| D8 | 2 - D3 |
| D9(PWM) | 2 - D2 |

Таблица 9 – подключение кнопок.

|  |  |
| --- | --- |
| Пин микроконтроллера | Действие кнопки |
| D0 | Назад |
| D1 | Вперед |

* 1. Компьютерная симуляция прототипа устройства и создание электросхемы.

Перед сборкой реального прототипа была проведена симуляция в онлайн редакторе AUTODESK Tinkercad, основанная на данных из составленных таблиц. Функционал данного инструмента позволяет собрать наглядный виртуальный прототип электрической цепи из произвольных компонентов с учетом подключения к различным микроконтроллерам. После создания такого прототипа для предполагаемого устройства (Приложение 1) была проведена симуляция, проверяющая цепь на короткие замыкания и возможные другие ошибки при соединении различных электрических компонентов. Произведя отладку с помощью данного инструмента и убедившись в надежности схемы, при помощи встроенной функции Tinkercad была произведена генерация графической электросхемы (Приложение 2). Так же стоит отметить, что данный онлайн инструмент предоставляет возможность симуляции загрузки написанного для микроконтроллера кода, однако в рамках данной работы эта часть его функционала почти не была использована, ввиду механической специфики задачи по моторизации потенциометров.

* 1. Моделирование корпуса устройства

Для моделирования корпуса был использован инструмент КОМПАС-3D. Сам корпус состоит из основного каркаса длиной и шириной в 190 мм, с различной высотой у задней и передней части – 50 мм и 30 мм соответственно (Приложение 3). Такой перепад создает удобный для руки наклон, направленный в сторону пользователя. Верхняя крышка является съемной, и содержит в себе симметрично расположенные продольные вырезы для потенциометров длинной 70 мм и шириной 5 мм. Так же сверху на ней располагается 2 отверстия диаметром 12,5 мм для кнопок переключения группы каналов. С задней стороны были оставлены отверстия для подключения микроконтроллера при помощи USB 3.0 type C и внешнего блока питания со штекером DC 5,5 – 2,5. Также были сделаны модели шкивов со внешним диметром в 10 мм и внутренним в 2 мм для направляющих потенциометр резиновых ремней. Последней смоделированной деталью является прямоугольная платформа 130х30 мм, предназначенная для крепления на нее потенциометра и соответствующего ему мотора, с предусмотренными отверстиями для использования типовых винтов диаметром 2 мм и 3 мм соответственно. Платформы с моторизованными потенциометрами крепятся под основной крышкой на специально предусмотренные для этого пластиковые ноги. Печать шкивов предусматривается в количестве 8 штук, а печать платформ в количестве 4 штук.

* 1. Создание модели печатной платы.

Для проектирования печатной платы использовался онлайн инструмент EasyEDA. Были учтены габариты смоделированного корпуса устройства и полученная ранее электросхема. При помощи встроенной функции была произведена автоматическая разводка линий электропитания по плате, после чего были внесены ручные коррективы, с учетом внутреннего расположения компонентов и удобства их пайки. Полученная плата (Приложение 4) предполагает необходимость соединения с помощью проводов только между платой и компонентами, находящимися у верхней панели устройства.

* 1. Построение UML диаграммы последовательности и блок схем.

С точки зрения архитектуры ПО, данное устройство состоит из программы, загружаемой на сам микроконтроллер. И скрипта, встраиваемого в DAW, интерпретирующего получаемые от устройства сообщения. Большинство современных цифровых рабочих аудио станций поддерживают работу с пользовательскими скриптами для микроконтроллеров. В рамках данной работы будет рассматриваться взаимодействие с FL Studio 21, как с наиболее популярной и наглядной для демонстрации функционала предполагаемого устройства. Для интеграции своих MIDI контроллеров их производители пишут скрипты на языке программирования Python, встраиваемые в эту DAW либо по умолчанию, либо вручную пользователем, если такое поддержка устройства по умолчание не предполагалась компанией Image Line (разработчик серии DAW FL Studio). Таким образом весь необходимый к написанию код можно представить в виде двух блок-схем, отображающих изолированные алгоритмы работы устройства и скрипта-обработчика, а также одной UML диаграммы последовательности, отображающей последовательность их коммуникации между собой при помощи протокола MIDI (Приложения 5-7).

* 1. Использование протокола MIDI.

MIDI (Musical Instrument Digital Interface) — цифровой интерфейс музыкальных инструментов. Данный протокол широко используется для коммуникаций между различными цифровыми аудио инструментами между собой и ПК. Общение происходит при помощи пакетов данных, содержащих в себе 3 байта. Первый называется байтом статуса, может принимать значения от 0 до 255 и содержит в себе согласно официальной спецификации сумму номера команды и номера канала, который в свою очередь не может превышать 16. Второй и третий байт не имеют собственного названия и несут в себе контекстную информацию в зависимости от канала, или не используются в принципе. Однако от использования официальной спецификации пришлось сразу отказаться, ввиду того что современные DAW поддерживают существенно большее количество каналов, чем 16. Потому для коммуникации для данного устройства были переопределены функции для двух типов MIDI сообщений (Таблица 10).

Таблица 10 – переопределенные MIDI сообщения.

|  |  |  |  |
| --- | --- | --- | --- |
| Название команды | Байт 1 | Байт 2 | Байт 3 |
| Смена громкости | 176 | 0-3  (Номер канала на выбранной группе каналов) | 0-127  (Громкость) |
| Смена группы каналов | 144 | 0-127  (Номер группы каналов) | Не используется |

В данной главе были рассмотрены все теоретические аспекты, входящие в планирование разработки MIDI контроллера, предназначенного для регулировки уровней громкости, на основе микроконтроллера типа Arduino. В следующей главе будет рассматривать процесс сборки, программирования и отладки устройства.

Глава 2. Практическая часть.

2.1 Программирование микроконтроллера Arduino.

В данной главе, практическая реализация этапов создания устройства рассмотрена в обратном порядке относительно первой главы. Это обусловлено несколькими причинами. Первой является время изготовления и доставки всех необходимых компонентов, ведь в рамках выбранной стратегии минимизации стоимости устройства в производстве это занимает от двух недель до месяца для разных компонентов. Второй причиной является безопасность. При создании электронного устройства принято для начала собирать все участки его цепи отдельно и сразу после этого вместе на беспаечной макетной плате. Неверно запрограммированный микроконтроллер может повредить купленные для финальной сборки компоненты, в том числе и самого себя. По этой причине практическая глава дипломной работы начинается с написание программы для микроконтроллера Arduino.

Разработка велась в официальной интегрированной среде разработки Arduino IDE. Несмотря на меньшую степень удобства по сравнению с другими редакторами кода по таким критериям как авто дополнение кода и подсветка синтаксиса, Arduino IDE предоставляет встроенное распознавание подключённого микроконтроллера и поддержку загрузки написанного кода на него при помощи одной кнопки, не выходя из рабочей среды. Что заметно экономит время при итеративной разработке, ввиду отсутствия ошибок совместимости, присущих, например, VS Code с установленным на нее плагином для работы с микроконтроллерами такого типа. Еще одним характерным преимуществом данной среды является менеджер установленных библиотек. На многих языках программирования такой функционал не добавлял бы ничего нового, однако программирование микроконтроллеров (не только типа Arduino) преимущественно происходит с использованием языков программирования C\C++, у которых нет встроенного менеджера библиотек.

Файловая структура проекта может быть произвольной за исключением двух обязательных требований. Корневой каталог должен содержать файл с расширением «.ino» и должен иметь одинаковое с ним название. Внутри файлы .ino содержат в себе упрощенный синтаксис языка программирования C++. Основные упрощения состоят в отсутствии ручного менеджмента пространств имен для импортированных модулей, в отсутствии необходимости ручного импортирования других файлов с расширением «.ino», находящихся в той же директории и в отсутствии необходимости импортирования стандартной библиотеки Arduino, предоставляющей готовые функции и удобные константы для работы с вводом и выводом микроконтроллера. Однако все эти упрощения существуют только на уровне содержания файла – каждый раз при сборке и компиляции проекта, специальный текстовый предпроцессор обрабатывает все содержимое рабочей директории, конкатенируя все «.ino» файлы в один, подставляя нужные пространства имен и импортируя стандартную библиотеку. После чего компиляция происходит на чистом C++.

Основной файл, необходимый для запуска проекта, должен содержать в себе две функции – setup() и loop(). В первой можно прописать логику инициализации программы, происходящей каждый раз при запуске устройства. Вторая функция содержит в себе основную логику программы, которая будет повторятся в бесконечном цикле при включенном состоянии микроконтроллера.

В первую очередь была разработана часть программы, отвечающая за работу с моторизованным потенциометром. Для удобства абстракция отражающая физический объект потенциометра была реализована с использованием объектно ориентированного программирования и инкапсулирована в свой класс под названием Fader.

Данный класс при инициализации принимает на вход номер пина микроконтроллера, соответствующего аналоговому пину для чтения значения потенциометра, номер цифрового пина для задания направления вращения, соответствующего данному потенциометру мотора и номер цифрового пина, способного воспроизводить широтно импульсно модулированный сигнал для регулирования скорости вращения мотора. Внутри класс содержит приватные переменные для хранения информации о состоянии потенциометра, и публичные методы для ее получения и изменения положения ручки потенциометра. Так же стоит отметить важный флаг isChangedFromDaw, который позволяет отслеживать источник изменений в положении потенциометра, и на основании этого принимать решение о том, надо ли отправлять соответствующее MIDI сообщение.

При написании этого фрагмента кода была обнаружена невозможность вынесения класса в свой собственный файл с расширением «.ino» из-за особенностей текстового предпроцессора для файлов данного типа. Она заключается в том, что любые файлы, не являющиеся основным для проекта, размещаются в коде конкатенированного общего файла после функций setup и loop, из-за происходит ошибка компиляции, вызванная неизбежным обращением к классу до его объявления. Однако данная проблема не является критической, ведь Arduino IDE оставляет за пользователем возможность написания собственных импортируемых файлов и библиотек на чистом C++, в которых возможно использовать стандартную библиотеки для работы с Arduino. Таким образом созданный класс был объявлен в файле заголовка Fader.h

#pragma once

#include <stdint.h>

#include <Arduino.h>

#define MOTOR\_SPEED 255

#define FORWARD 0

#define BACKWARD 1

#define STOP 2

#define ANALOG\_VALUE\_THRESHOLD 8

#define MIDI\_VALUE\_THRESHOLD 2

#define DEFAULT\_MIN\_VALUE 0

#define DEFAULT\_MAX\_VALUE 1023

#define STUCK\_THRESHOLD 10

class Fader {

  private:

    bool isChangedFromDaw;

    int prevAnalogValue, currentAnalogValue, potentiometrMinValue, potentiometrMaxValue;

    uint8\_t potentiometrPin, motorInPin, motorPWMPin, currentMidiValue, prevMidiValue;

    int convertMidiToAnalogValue(uint8\_t midiValue);

    uint8\_t convertAnalogToMidiValue(int analogValue);

    void updateCurrentValue();

    void waitForPotentiometrBound();

    void controlMotor(uint8\_t direction);

  public:

    Fader(uint8\_t potentiometrPin, uint8\_t motorInPin, uint8\_t motorPWMPin);

    void setPosition(uint8\_t midiValue);

    uint8\_t getCurrentValue();

    bool getIsChangedFromDaw();

    void setIsChangedFromDaw(bool isChangedFromDaw);

};

И реализован в файле Fader.cpp

#include "Fader.h"

Fader::Fader(uint8\_t potentiometrPin, uint8\_t motorInPin, uint8\_t motorPWMPin) {

  this->potentiometrPin = potentiometrPin;

  this->motorInPin = motorInPin;

  this->motorPWMPin = motorPWMPin;

  this->potentiometrMinValue = DEFAULT\_MIN\_VALUE;

  this->potentiometrMaxValue = DEFAULT\_MAX\_VALUE;

  this->prevAnalogValue = DEFAULT\_MAX\_VALUE + ANALOG\_VALUE\_THRESHOLD;

  this->prevMidiValue = 128;

  this->isChangedFromDaw = false;

  pinMode(motorInPin, OUTPUT);

  pinMode(motorPWMPin, OUTPUT);

  this->updateCurrentValue();

}

int Fader::convertMidiToAnalogValue(uint8\_t midiValue){

  int analogValue = map(midiValue, 0, 127, this->potentiometrMinValue, this->potentiometrMaxValue);

  return analogValue;

}

uint8\_t Fader::convertAnalogToMidiValue(int analogValue){

  int midiValue = map(analogValue, this->potentiometrMinValue, this->potentiometrMaxValue, 0, 127);

  return midiValue;

}

void Fader::updateCurrentValue(){

  this->currentAnalogValue = analogRead(this->potentiometrPin);

  if (abs(this->currentAnalogValue - this->prevAnalogValue) >= ANALOG\_VALUE\_THRESHOLD){

    this->prevMidiValue = this->currentMidiValue;

    this->currentMidiValue = this->convertAnalogToMidiValue(this->currentAnalogValue);

    this->prevAnalogValue = this->currentAnalogValue;

  }

}

void Fader::controlMotor(uint8\_t direction){

  uint8\_t speed = MOTOR\_SPEED;

  switch (direction) {

    case BACKWARD:

      speed = 0;

      break;

    case STOP:

      speed = 0;

      direction = FORWARD;

      break;

  }

  analogWrite(this->motorPWMPin, speed);

  digitalWrite(this->motorInPin, direction);

}

uint8\_t Fader::getCurrentValue(){

  this->updateCurrentValue();

  return this->currentMidiValue;

};

void Fader::setPosition(uint8\_t midiValue){

  this->updateCurrentValue();

  int difference = this->currentMidiValue - midiValue;

  if (!difference) return;

  uint8\_t direction = 0 + (difference < 0);

  int sign = (difference > 0) - (difference < 0);

  this->controlMotor(direction);

  int stuckCounter = 0;

  while ((abs(this->currentMidiValue - midiValue - sign \* MIDI\_VALUE\_THRESHOLD) >= MIDI\_VALUE\_THRESHOLD) && stuckCounter <= STUCK\_THRESHOLD){

    this->updateCurrentValue();

    bool isStuckAtTheBottom = (abs(0 - this->currentMidiValue) <= MIDI\_VALUE\_THRESHOLD) && (abs(0 - midiValue) <= MIDI\_VALUE\_THRESHOLD);

    bool isStuckAtTheTop = (abs(127 - this->currentMidiValue) <= MIDI\_VALUE\_THRESHOLD) && (abs(127 - midiValue) <= MIDI\_VALUE\_THRESHOLD);

    if (isStuckAtTheBottom || isStuckAtTheTop) stuckCounter++;

    delay(1);

  }

  this->controlMotor(STOP);

}

bool Fader::getIsChangedFromDaw(){

  return this->isChangedFromDaw;

}

void Fader::setIsChangedFromDaw(bool isChangedFromDaw){

  this->isChangedFromDaw = isChangedFromDaw;

}

Для одновременной работы с четырьмя потенциометрами был написан класс FaderManager, хранящий в себе после инициализации объекты класса Fader, информацию об их предыдущих и следующих значениях, а так же предоставляющий публичные методы для получения и изменения их состояния.

{Код}

Работа с кнопками так же осуществлена через написание собственного класса. Класс Button при инициализации получает на вход номер пина микроконтроллера, к которому подключена кнопка, и позволяет получить информацию о том, была ли она нажата. При этом нажатие срабатывает только один раз за шаг жизненного цикла программы, предотвращая эффект множественного нажатия.

{Код}

Отправка MIDI сообщений реализована при помощи сторонней библиотеки MIDIUSB, и организована в виде двух функций в файле midi.ino. Данная библиотека позволяет автоматически без дополнительной настройки подключать микроконтроллер Arduino к ПК как устройство класса MIDI. Написанные функции согласно названию отвечают за получение и отправку сообщений.

{Код}

Для хранения константных значений пинов и кодов статуса MIDI сообщений были созданные файлы Pins.h и MessageStatus.h. Это позволило получить к ним доступ из любого места программы и визуально разгрузило основной файл.

#define BUTTON\_PREV\_LAYER\_PIN 0

#define BUTTON\_NEXT\_LAYER\_PIN 1

#define POTENTIOMETR\_PIN\_1 A0

#define MOTOR\_IN\_PIN\_1 2

#define MOTOR\_PWM\_PIN\_1 3

#define POTENTIOMETR\_PIN\_2 A1

#define MOTOR\_IN\_PIN\_2 14

#define MOTOR\_PWM\_PIN\_2 5

#define POTENTIOMETR\_PIN\_3 A2

#define MOTOR\_IN\_PIN\_3 7

#define MOTOR\_PWM\_PIN\_3 6

#define POTENTIOMETR\_PIN\_4 A3

#define MOTOR\_IN\_PIN\_4 15

#define MOTOR\_PWM\_PIN\_4 9

Pins.h

#define VOLUME\_CHANGE  0xB0

#define LAYER\_CHANGE 0x90

MessageStatus.h

Все написанные классы и функции используются в основном файле midiController.ino. При инициализации программы происходит инициализация глобальных переменных менеджера потенциометров и кнопок. Каждый шаг жизненного цикла программа для начала проверяет полученные MIDI сообщения и нажатие кнопок, после чего либо считывает и отправляет значение потенциометров, либо выставляет его согласно полученному с ПК.

{Код}

2.2 Написание скрипта обработчика для DAW FL Studio 21.

Как уже было сказано ранее скрипты для обработки MIDI сообщений в FL Studio пишутся с использованием языка программирования Python. Данная DAW предоставляет свое API в виде библиотеки для этого языка. Логика написания скрипта состоит в объявлении функций с предустановленными названиями, которые будут автоматически вызываться программой в случае соответствующих им событий, таких как изменение громкости канала. В ходе работы был использован редактор кода VS Code с настроенным виртуальным окружением, содержащим стороннюю библиотеку fl\_classes, которая обеспечивает подсветку синтаксиса и удобную навигацию по API. Структура проекта помимо предустановленных названий для функций обратного вызова, требует, чтобы основной файл с расширением «.py», находящийся по пути «...Documents\Image-Line\FL Studio\Settings\Hardware\devicename\device\_devicename.py» на ОС Windows обязательно имел закомментированную первую строку вида «# name=Fader Control Surface», содержащую в себе название, которое в последствии будет использовано в DAW при выборе типа устройства. Название основного файла обязано начинаться с «device\_». Корневой каталог содержащий скрипт может иметь произвольное название, так же произвольной является часть названия скрипта после обязательной.

С точки зрения архитектуры и организации кода было реализовано похожее на программу для микроконтроллера решение. В первую очередь был написан класс Track, инкапсулирующий в себе всю важную информацию об отдельно взятом канале микшера в контексте его использования с MIDI устройством. Так же он имеет два служебных метода для конвертации MIDI значения громкости в число с плавающей точкой от нуля до единицы и обратно, и публичный метод для сброса состояния флагов, сигнализирующих о факте изменения состояния канала и источнике этого изменения.

import mixer

class Track():

    def \_\_init\_\_(self, device\_number: int, daw\_number: int):

        self.\_device\_number = device\_number

        self.\_daw\_number = daw\_number

        self.\_is\_changed = False

        self.\_is\_changed\_from\_device = False

    @property

    def device\_number(self) -> int:

        return self.\_device\_number

    @property

    def daw\_number(self) -> int:

        return self.\_daw\_number

    @daw\_number.setter

    def daw\_number(self, daw\_number: int):

        self.\_daw\_number = daw\_number

    @property

    def is\_changed(self) -> bool:

        return self.\_is\_changed

    @is\_changed.setter

    def is\_changed(self, is\_changed: bool):

        self.\_is\_changed = is\_changed

    @property

    def is\_changed\_from\_device(self) -> bool:

        return self.\_is\_changed\_from\_device

    @is\_changed\_from\_device.setter

    def is\_changed\_from\_device(self, is\_changed\_from\_device: bool):

        self.\_is\_changed\_from\_device = is\_changed\_from\_device

    @property

    def volume(self) -> int:

        return self.\_convert\_float\_to\_midi(mixer.getTrackVolume(self.daw\_number))

    @volume.setter

    def volume(self, volume: int):

        mixer.setTrackVolume(self.daw\_number, self.\_convert\_midi\_to\_float(volume))

    def \_convert\_midi\_to\_float(self, midiValue: int) -> float:

        return int(midiValue) / 127

    def \_convert\_float\_to\_midi(self, floatValue: float) -> int:

        return round(floatValue \* 127)

    def reset\_state(self):

        self.is\_changed = False

        self.is\_changed\_from\_device = False

Далее был реализован класс менеджер под названием TrackManager, позволяющий отслеживать изменения в текущей выбранной группе каналов и получать их состояния.

from track import Track

class TrackManager():

    def \_\_init\_\_(self, num\_tracks: int, current\_group: int):

        self.\_num\_tracks = num\_tracks

        self.\_current\_group = current\_group

        self.\_tracks = [Track(device\_number, self.\_get\_daw\_track\_number(device\_number)) for device\_number in range(0, self.\_num\_tracks)]

    @property

    def num\_tracks(self) -> int:

        return self.\_num\_tracks

    @property

    def current\_group(self) -> int:

        return self.\_current\_group

    @current\_group.setter

    def current\_group(self, current\_group: int):

        self.\_current\_group = current\_group

        for track in self.\_tracks:

            track.reset\_state()

            track.daw\_number = self.\_get\_daw\_track\_number(track.device\_number)

    @property

    def tracks(self) -> list:

        return self.\_tracks

    @property

    def changed\_from\_daw\_tracks(self) -> list:

        return [track for track in self.\_tracks if track.is\_changed and not track.is\_changed\_from\_device]

    @property

    def changed\_from\_device\_tracks(self) -> list:

        return [track for track in self.\_tracks if track.is\_changed and track.is\_changed\_from\_device]

    def get\_track\_by\_daw\_number(self, daw\_number: int) -> Track | None:

        search\_result = list(filter(lambda track: track.daw\_number == daw\_number, self.\_tracks))

        if len(search\_result):

            return search\_result[0]

        else:

            return None

    def get\_track\_by\_device\_number(self, device\_number: int) -> Track | None:

        search\_result = list(filter(lambda track: track.device\_number == device\_number, self.\_tracks))

        if len(search\_result):

            return search\_result[0]

        else:

            return None

    def \_get\_device\_track\_number(self, daw\_track\_number: int) -> Track | None:

        device\_track\_number = daw\_track\_number - self.\_current\_group \* self.\_num\_tracks - 1

        if device\_track\_number >= 0 and device\_track\_number < self.\_num\_tracks:

            return device\_track\_number

        else:

            return None

    def \_get\_daw\_track\_number(self, device\_track\_number: int) -> int:

        return device\_track\_number + 1 + self.\_current\_group \* self.\_num\_tracks

    def reset\_track\_state(self):

        for track in self.\_tracks:

            track.reset\_state()

В файле midi\_services.py была организована работа с отправкой midi сообщений. Функция send\_midi отвечает за отправку одиночного сообщения, в то время как функция send\_current\_group\_volume отправляет 4 сообщения подряд с информацией о громкости всех каналов выбранной группы. Она используется при инициализации скрипта и при смене активной группы.

import device

from constants import \*

from track\_manager import TrackManager

def send\_midi(status: int, data1: int, data2: int):

    message = status + (data1 << 8) + (data2 << 16)

    device.midiOutMsg(message)

    if DEBUG:

        print(f"Sent: {status}-{data1}-{data2}")

def send\_current\_group\_volume(track\_manager: TrackManager):

    for track in track\_manager.tracks:

        send\_midi(VOLUME\_CHANGE\_STATUS\_CODE, track.device\_number, track.volume)

В файле handlers.py реализованы обработчики для двух переопределенных в рамках дипломной работы типов MIDI сообщений.

import mixer

from track\_manager import TrackManager

from midi\_services import send\_current\_group\_volume

def volume\_change\_handler(event, track\_manager: TrackManager):

    device\_track\_number = event.data1

    volume = event.data2

    track = track\_manager.get\_track\_by\_device\_number(device\_track\_number)

    track.is\_changed = True

    track.is\_changed\_from\_device = True

    track.volume = volume

def group\_change\_handler(event, track\_manager: TrackManager):

    group\_number = event.data1

    if group\_number >= 0 and group\_number <= 31 and group\_number != track\_manager.current\_group:

        group\_first\_track = track\_manager.get\_track\_by\_device\_number(0)

        track\_manager.current\_group = group\_number

        mixer.setActiveTrack(group\_first\_track.daw\_number)

        send\_current\_group\_volume(track\_manager)

В файле constants.py вынесены константные значения параметров запуска, режима отладки и переопределённых кодов сообщений.

DEBUG = True

NUM\_TRACKS = 4

DEFAULT\_CURRENT\_GROUP = 0

VOLUME\_CHANGE\_STATUS\_CODE = 176

GROUP\_CHANGE\_STATUS\_CODE = 144

В основном файле device\_FaderControlSurface.py находится реализация функций обратного вызова, написанных согласно официальной документации FL Studio API. Функция OnInit вызывается при инициализации скрипта, которая автоматически происходит при распознании устройства внутри FL Studio. В ней выполняется инициализация менеджера каналов и словаря, описывающего соответствие кодов сообщений и их обработчика. Функция OnMidiIn вызывается первой при любом входящем MIDI сообщении, и используется для фильтрации не определенных типов сообщений. После для всех известных типов сообщений вызывается функция OnMidiMsg, которая запускает соответствующий для сообщения обработчик. Функция OnDirtyMixerTrack вызывается по событию изменения громкости любого из каналов, в тот момент, когда новое значение громкости еще неизвестно. Она используется для сбора информации об изменениях. После нее вызывается функция OnRefresh, которая срабатывает от ряда различных событий внутри DAW, однако в данной реализации все события, не относящиеся к изменению громкости каналов не отфильтровываются, после чего информация уже обрабатывается информация об измененных каналах, получается их значение громкости и отправляется в виде MIDI сообщщения на контроллер.

# name=Fader Control Surface

import mixer

import midi

from constants import \*

from track\_manager import TrackManager

from midi\_services import send\_midi, send\_current\_group\_volume

from handlers import volume\_change\_handler, group\_change\_handler

track\_manager: TrackManager = None

midi\_message\_handlers: dict = None

def OnInit():

    global track\_manager

    global midi\_message\_handlers

    track\_manager = TrackManager(NUM\_TRACKS, DEFAULT\_CURRENT\_GROUP)

    midi\_message\_handlers = {

        VOLUME\_CHANGE\_STATUS\_CODE: volume\_change\_handler,

        GROUP\_CHANGE\_STATUS\_CODE: group\_change\_handler

    }

    mixer.setActiveTrack(track\_manager.get\_track\_by\_device\_number(0).daw\_number)

    send\_current\_group\_volume(track\_manager)

    if DEBUG:

        print("")

        print("MIDI messages handlers:")

        for message in midi\_message\_handlers:

            print(f"status: {message}, handler: {midi\_message\_handlers[message]}")

        print("")

        print("Manager initial state:")

        for track in track\_manager.tracks:

            print(track.\_\_dict\_\_)

def OnMidiIn(event):

    if DEBUG:

        print(f"Received: {event.status}-{event.data1}-{event.data2}")

    if int(event.status) not in list(midi\_message\_handlers.keys()):

        event.handled = True

def OnMidiMsg(event):

    midi\_message\_handlers[event.status](event, track\_manager)

    event.handled = True

def OnDirtyMixerTrack(daw\_track\_number):

    if daw\_track\_number == -1:

        return

    track = track\_manager.get\_track\_by\_daw\_number(daw\_track\_number)

    if track and not track.is\_changed:

        track.is\_changed = True

        track.is\_changed\_from\_device = False

def OnRefresh(flag):

    if flag != midi.HW\_Dirty\_Mixer\_Controls:

        return

    for track in track\_manager.changed\_from\_daw\_tracks:

        send\_midi(VOLUME\_CHANGE\_STATUS\_CODE, track.device\_number, track.volume)

    track\_manager.reset\_track\_state()

2.3 Конфигурация устройства в DAW.

После загрузки скрипта и подключения устройства было необходимо установить их соответствие между собой. Это было сделано в меню Options > MIDI settings. При подключении устройства в списке MIDI устройств автоматически появляется новая строка с подписью «Arduino micro». После клика на нее в контекстном меню «Controller type» была выбрана опция FaderControlSurface (Приложение 5), являющаяся написанным ранее скриптом.

2.4 Сборка финальной версии устройства.

После тестирования полученного кода на тестовом макете на предмет безопасности и отказоустойчивости, была произведена сборка окончательной версии устройства.

В первую очередь из всех потенциометров были извлечены металлические шарики, предназначенные для фиксации в центральном положении. Далее на плату был припаян микроконтроллер и разъем для питания моторов. После чего плата была размещена на дне основной части корпуса и закреплена винтами. Далее на промежуточную между дном и крышкой плоскость были установлены потенциометры, моторы и кнопки и так же закреплены винтами. После чего при помощи проводов они были припаяны к соответствующим контактам на плате. Было произведено тестовое подключение с целью выявить плохо спаянные контакты и работоспособность платы в целом. После отладки элемент был установлен на поддерживающие платформы и закреплен винтами. После чего поверх была установлена и закреплена крышка, а также колпаки на потенциометры. Собранное устройство прошло ряд тестов, которые не выявили неисправностей и ошибок в работе.