Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Разработка ПО для поворотного стенда на базе встроенного микрокомпьютера

Выполнил студент Семикозов Я.Л.

гр. 3331506/10401

Преподаватель Ананьевский М. С.

Санкт-Петербург
2024

Оглавление

1. Введение

Необходимо разработать программное обеспечение для управления поворотным стендом на базе микрокомпьютера с unix-подобной ОС. Исходя из поставленной задачи, можно выделить 4 основные подзадачи в курсовой работе:

1. Написать программный модуль для коммуникации микрокомпьютера и управляющего центра (любой ПК с необходимыми компонентами среды) посредством сетевого сокета TCP/UDP протокола. Описать протокол коммуникации;
2. Написать программный модуль для сбора и первичной обработки данных с аналого-цифрового преобразователя ADS1115;
3. Написать программный модуль для отправки http-запросов на локальный хост управляющей;
4. Скомпоновать программные модули в единую программу.

1.1. Среда выполнения и используемые технологии

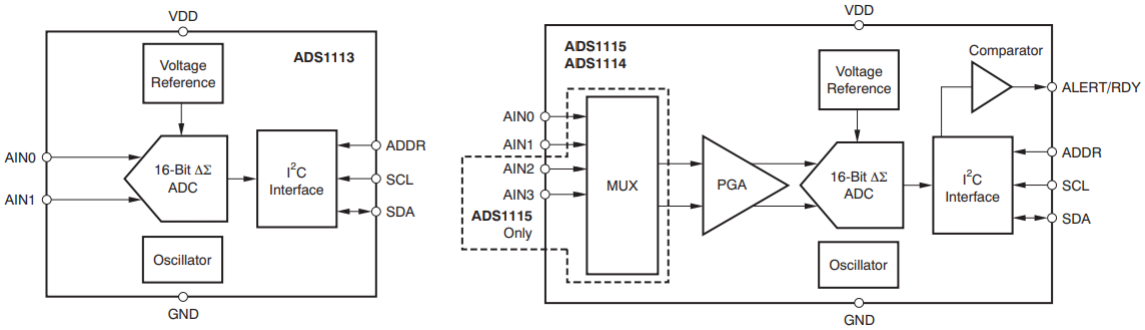
За отсутствием жестких требований было решено, что в качестве среды разработки эффективнее всего использовать редактор кода Visual Studio Code, так как все необходимые инструменты к моменту разработки были уставновлены.

Сборка проекта включила в себя: код программных модулей, собранный в отдельные статические библиотеки и связующий код основной программы. В конечном итоге собранный проект компоновался из собственного исходного кода и 3 собранных библиотек, решающих соответствующие подзадачи.

Стек проекта включает в себя систему сборки – CMake 3.25, библиотеку стандартных шаблонов **STL** (версия стандарта C++ – 20), модуль библиотеки **boost** asio для работы с асинхронной отправкой данных по сетевому сокету.

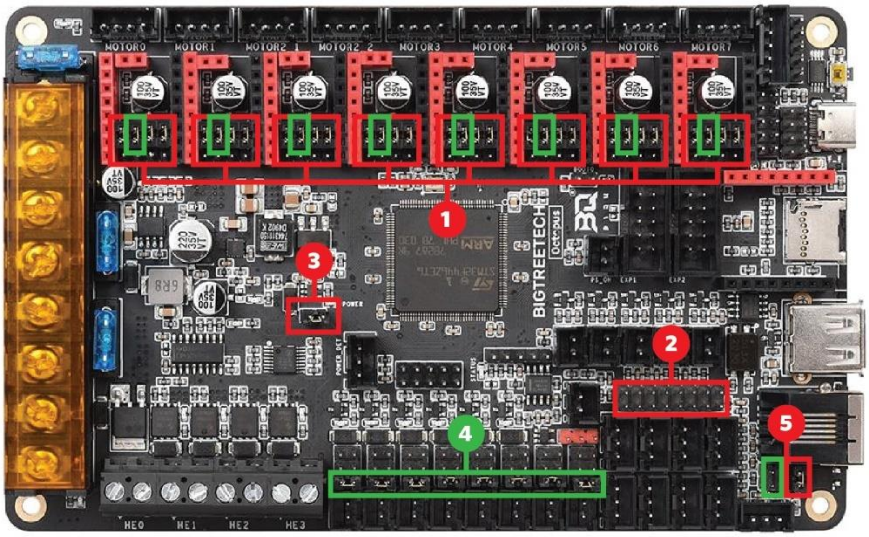
2. Описание проекта

2.1. Аналого-цифровой преобразователь ADS1115



Датчик ADS1115 - 16-ти разрядный аналогово-цифровой преобразователь с 4 выходами для преобразования аналогового сигнала в цифровой. Датчик ADS1115 разработанный на базе одноименного чипа, маломощного, 16-разрядного, совместимый с I2C шиной. Оснащен программируемым усилителем и цифровым компаратором. Выполняет преобразования со скоростью передачи данных от 8 до 860 выборок в секунду (SPS). Имеет 4 входа для преобразования аналогового сигнала. Диапазоны входных сигналов от ± 256 мВ до ± 6.144 В. Питание модуля толерантно к 3,3 В или 5 В.

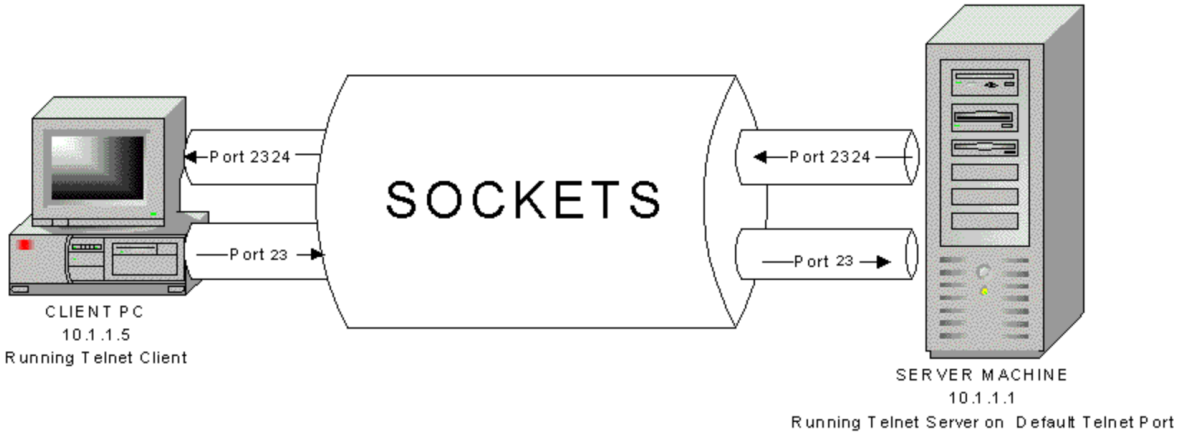
2.2. Управление поворотным стендом



BigTreeTech Octopus - это мощная и многофункциональная плата управления для 3D принтера. В общей сложности плата поддерживает до 8 драйверов для шаговых двигателей и до 9 шаговый двигателей. На борту используется 32-битный основной управляющий чип ARM Cortex-M4 серии STM32F446ZET6 с частотой ядра 180 МГц.

Основной критерий выбора – поддержка работы прошивки Клиппер. Поддержка Raspberry Pi с использованием эмулированного последовательного порта через USB или прямого подключения UART через любой из последовательных портов на плате.

2.3. Сетевые сокеты



Сокет — это тип сетевого подключения, устанавливаемого между двумя компьютерными процессами. Как правило, процессы выполняются на двух разных компьютерах, подключенных к IP-сети. Однако подключенные процессы могут выполняться на одном компьютере при использовании IP-адреса «локального хоста». Он является чем-то вроде "портала", через которое можно отправлять байты во внешний мир. Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.

Сокет TCP - это сокет, ориентированный на подключение, который использует протокол управления передачей (TCP). Для установки соединения требуется три пакета: SYN-пакет, SYN-ACK-пакет и ACK-пакет подтверждения. TCP-сокет определяется IP-адресом компьютера и используемым им портом. TCP-сокет гарантирует получение и подтверждение всех данных.

Например, мы отправляем HTTP-запрос от нашего клиента по адресу 120.1.1.1 на веб-сайт по адресу 189.1.1.1. Сервер для этого веб-сайта будет использовать хорошо известный номер порта 80, поэтому его сокет равен 189.1.1.1:80, как мы видели ранее. у нас был временный номер порта 3022 для веб-браузера, поэтому клиентский сокет - 120.1.1.1:3022. Общее соединение между этими устройствами можно описать с помощью этой пары сокетов: (189.1.1.1:80, 120.1.1.1:3022).

Концепция сокетов

Сетевой сокет во многом напоминает электрическую розетку. В сети имеется множество сокетов, причем каждый из них выполняет стандартные функции. Все, что поддерживает стандартный протокол, можно «подключить» к сокету и использовать для коммуникаций. Для электрической розетки не имеет значения, что именно вы подключаете – лампу или тостер, поскольку оба прибора рассчитаны на напряжение 220 Вольт и частоту 50 Герц. Несмотря на то, что электричество свободно распространяется по сети, все розетки в доме имеют определенное место.

Подобным образом работают и сетевые сокеты, за исключением того, что электроны и почтовые адреса заменены на пакеты **TCP/IP** и IP-адреса. Internet Protocol (IP) является низкоуровневым протоколом маршрутизации, который разбивает данные на небольшие пакеты и рассылает их по различным сетевым адресам, что не гарантирует доставку вышеупомянутого пакета адресату. Transmission Control Protocol (TCP) является протоколом более высокого уровня, собирающим пакеты в одну строку, сортирующим и перетранслирующим их по мере необходимости, поддерживая надежную рассылку данных. Третий протокол, UNIX Domain Protocol (UDP), используется вместе с TCP и может применяться для быстрой, но ненадежной передачи пакетов.

Socket API был впервые реализован в операционной системе Berkley UNIX. Сейчас этот программный интерфейс доступен практически в любой модификации Unix, в том числе в Linux. Хотя все реализации чем-то отличаются друг от друга, основной набор функций в них совпадает. Изначально сокеты использовались в программах на C/C++, но в настоящее время средства для работы с ними предоставляют многие языки (Perl, Java и др.).

Сокеты предоставляют весьма мощный и гибкий механизм межпроцессного взаимодействия (**IPC**). Они могут использоваться для организации взаимодействия программ на одном компьютере, по локальной сети или через Internet, что позволяет вам создавать распределённые приложения различной сложности. Кроме того, с их помощью можно организовать взаимодействие с программами, работающими под управлением других операционных систем. Например, под Windows существует интерфейс Window Sockets, спроектированный на основе socket API. Ниже мы увидим, насколько легко можно адаптировать существующую Unix-программу для работы под Windows.

В настоящем проекте было решено использовать сетевой протокол TCP, т.к. данные должны отправляться без потерь.

3. Реализация

3.1. Общие концепции

Для единой структуры модулей был реализован класс WorkerBase, который является базовым классом для трех других классов в проекте. Его суть проста – предоставить общий интерфейс для работы с классами и упростить работу с потоками.

Основные причины использования интерфейсов

Абстракция: Интерфейсы позволяют создавать абстрактные классы, которые определяют контракт без реализации. Это позволяет разработчикам сосредоточиться на определении поведения без необходимости беспокоиться о реализации.
Полиморфизм: Классы могут наследовать от нескольких интерфейсов, предоставляя реализации для всех соответствующих методов.
Согласованность контракта: Использование интерфейсов обеспечивает ясные контракты между разными частями программы, улучшая читаемость и обеспечивая однородность в структуре приложения.
Уменьшение повторного использования кода: Интерфейсы позволяют повторно использовать код, реализующий общую структуру класса, в разных частях программы.
Инкапсуляция: Интерфейсы могут использоваться для инкапсуляции поведения, скрывая внутренние детали реализации и упрощая интерфейс для клиентов уровнем выше.

```
using ID = Message::Instance;

class WorkerBase
{
public:
    WorkerBase() = delete;
    WorkerBase(ID first, ID second = ID::None) : first(first), second(second) {}

    virtual ~WorkerBase()
    {
        if (!t) { return; }
        enable = false;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        if (t->joinable()) { t->join(); }
        delete t;
    }

    void start()
    {
```

```
        if (t) { return; }
        this->prestart();
        t = new std::thread(&WorkerBase::run, this);
        enable = true;
    }

    [[maybe_unused]] inline void resume()
    {
        m.unlock();
    }

    [[maybe_unused]] inline void stop()
    {
        m.lock();
    }

    [[maybe_unused]] inline bool started() __attribute__((warn_unused_result))
    {
        return enable;
    }

    [[maybe_unused]] virtual void prestart() = 0;

    virtual void handle(std::unique_ptr<Message::ClientData> &payload) = 0;
    virtual void get(std::shared_ptr<std::uint8_t[]> &p) const = 0;

    inline void setBuffer(RTDataBuffer *b)
    {
        buffer = b;
    }
    inline bool at(ID id)
    {
        if ((id == this->first) || (id == this->second))
        {
            lastID = id;
            return true;
        }
        return false;
    }

protected:
    Message::Instance lastID;
    RTDataBuffer *buffer;
    std::atomic_bool enable{ false };
    std::thread *t{ nullptr };
    mutable std::mutex m;

protected:
    virtual void run() = 0;

private:
```

```
    ID first, second;
};
```

Данные-члены класса:

- lastID: Уникальный идентификатор последнего обработанного сообщения.
- buffer: Указатель на буфер данных реального времени.
- enable: Атомарная булева переменная, указывающая, включен ли поток исполнения.
- t: Указатель на поток исполнения.
- m: Взаимноисключающий мьютекс для синхронизации доступа к ресурсам.

Конструкторы:

- Конструктор по умолчанию (запрещен): Запрещает создание экземпляров WorkerBase без параметров.
- Параметризованный конструктор: Конструктор принимает два идентификатора сообщений и создает экземпляр класса.

Деструктор:

- Деструктор: Уничтожает поток исполнения и освобождает выделенную память.

Функции-члены класса:

- start(): Запускает поток исполнения для объекта.
- resume(): Разблокирует мьютекс, позволяя потоку продолжить выполнение.
- stop(): Блокирует мьютекс, приостанавливая поток исполнения.
- started(): Возвращает значение true, если поток исполнения запущен, и false в противном случае.
- get(p): Виртуальный метод, который возвращает данные, связанные с объектом.
- setBuffer(b): Устанавливает указатель на буфер данных реального времени.
- at(id): Проверяет, соответствует ли идентификатор сообщения одному из двух идентификаторов, связанных с объектом.

Виртуальные методы:

- run(): Реализуется производными классами и определяет поведение потока исполнения.
- prestart(): Реализуется производными классами и вызывается перед запуском потока исполнения.
- handle(payload): Виртуальный метод, который обрабатывает сообщение, переданное в качестве параметра.

Защищенные методы:

- enable: Управляет включением потока исполнения.

Приватные данные-члены:

- first, second: Идентификаторы сообщений, связанные с объектом.

Особенности:

- Класс обеспечивает базовый функционал для обработки сообщений и управления потоками исполнения.
- Наследники класса WorkerBase могут реализовать специфическую для домена логику в своих переопределениях виртуальных методов.
- Использование мьютекса гарантирует синхронизированный доступ к ресурсам, разделяемым между потоком исполнения и внешним кодом.

Структура протокола передачи данных между клиентом и сервером:

```
namespace Message
{
using HeaderType = uint32_t;

struct Common
{
    static constexpr HeaderType kDefaultHeader{0xABCDEFFF};

    HeaderType header{kDefaultHeader};
    Instance instance;
    Status err;

    uint32_t timeStamp;
} __attribute__((__packed__));

struct ClientData
{
    Command command;
    float main[2];
    uint8_t common;
} __attribute__((__packed__));

struct ServerData
{
    Query query;
    float adcRawValue[4];
```

```
    float sensorAngles[2];
    float standAngles[2];

    float voltage[4];

    float pointOfCenterMass[2];
    float pointOfStandAngles[2];
} __attribute__((__packed__));

struct ServerPacket
{
    Common common;
    ServerData load;
} __attribute__((__packed__));

struct ClientPacket
{
    Common common;
    ClientData load;
} __attribute__((__packed__));

}; // namespace Message
```

Каждая посылка, отправленная с серверной стороны, состоит из заголовка с типом посылки, типом класса-отправителя (необходимо для корректной обработки данных на обратной стороне) и статусом работы. Посылка, отправленная с клиентской стороны, включает в себя команду и дополнительное значение (если таковое имеется) команды.

Типы передаваемых посылок:

```
namespace Message
{

enum class Query : uint8_t
{
    StepperCompleted,
    ResponseDataRequest,
    Parked
};

enum class Instance : uint8_t
{
    // from host messages
    None = 0x00,
    Azimuth = 0xAA,
    Elevation = 0xBB,
    Both,
    ADS = 0xCC,
```

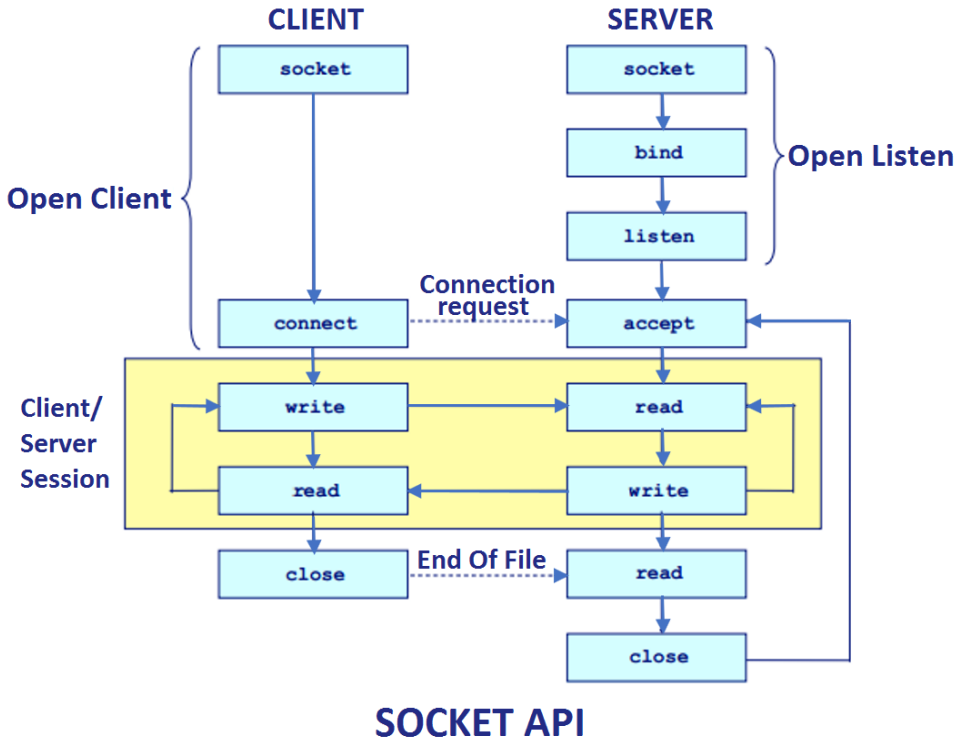
```
    Common = 0xDD,
    Error = 0xFA,
    END
};

enum class Status : uint32_t
{
    OK = 0,
    I2CError,
    USBError, // hard error
    StepperError,
    CommunicationError, // soft error
    UnAvailable
};
}; // namespace Message
```

Типы передаваемых команд:

```
namespace Commands
{

enum class ADC : uint8_t
{
    None = 0x01,

    SetFSR,
    SetSPS,
    SetMode,
    SetMux,

    GetFSR,
    GetSPS,
    GetMode,
    GetMux,

    SetRef, //!< change channel for channel, or channel for GND

    ReadMux,
    ReadAllMuxs,

    END //! NEEDED
};
```

```
enum class Stepper : uint8_t
{
    None = 0x21,

    DecreaseAngle,
    IncreaseAngle,

    ForceSetAngle, //!< set without move, needed for moving stepper on 90 degrees
after park
    SetAngle,
    SetAngles,
    SetSteps,

    SetRPM,
    Park,

    END //! NEEDED
};

enum class Common : uint8_t
{
    None = 0x41,

    GetCalibration,
    GetData,
    GetNoise,

    END //! NEEDED
};
}; // namespace Commands
```

3.2. Модуль работы с сетевыми сокетами



При написании модуля с сетевыми сокетами были поставлены следующие требования:

1. Если клиент решит завершить свою работу без отправки команды с уведомлением, сервер должен закрыть сокет и оставаться доступным для следующих подключений;
2. Отправки и прием должны быть асинхронным с основным выполняющим потоком программы, т.к. стенд может в это время проезжать по заданной траектории;
3. Сервер не должен обрабатывать никакие команды, его задача состоит в получении данных и дальнейшей передаче на уровень ниже.

Что такое boost.asio?

Если коротко, Boost.Asio это, большей частью, кросс-платформенная C++ библиотека для программирования сетей и некоторых других низкоуровневых программ ввода/вывода. Есть много реализаций для решения сетевых задач, но Boost.Asio превзошел их все; он был принят в Boost в 2005 и с тех пор был протестирован большим количеством пользователей. Boost.Asio успешно абстрагирует понятия input и output, которые работают не только для работы в сети, но и для последовательных COM-портов, файлов и так далее. Кроме этого вы можете делать input или output программирование синхронным или асинхронным. Собственно

последний факт и является причиной использования данного модуля, т.к. команды и ответы на команды могут приходить в разное время (издержки считывания данных с АЦП, отправка http-запроса на плату управления Octopus и тд).

```
namespace ByteStorm
{

using namespace boost::asio;
using namespace boost::posix_time;
using namespace boost::placeholders;

using Error = boost::system::error_code;
using Acceptor = ip::tcp::acceptor;
using Socket = ip::tcp::socket;
using Endpoint = ip::tcp::endpoint;

class ByteStormBoost;
class TCPConnection;

using Connection = boost::shared_ptr<TCPConnection>;
using Connections = std::vector<Connection>;

class TCPConnection : public boost::enable_shared_from_this<TCPConnection>,
                      boost::noncopyable,
                      public ByteStormBase<TCPConnection>
{
    friend class ByteStormBoost;

public:
    TCPConnection(io_service &service, size_t transfer_size,
ProcessorBase<TCPConnection> *h = nullptr);
    ~TCPConnection();

    void flush() override;

    Status send(std::unique_ptr<std::uint8_t[]> &data, const size_t size)
override;
    void read();
    void read_complete(const Error &err, size_t bytes_transferred);

    void write(std::unique_ptr<std::uint8_t[]> &data, const size_t size);
    void write_complete(const Error &err, size_t bytes_transferred);

    void on_connect();

    template <typename Fn, typename... Args>
```

```
    auto bind(Fn &&fn, Args &&...args)
    {
        return boost::bind(std::forward<Fn>(fn), shared_from_this(),
std::forward<Args>(args)...);
    }

private:
    void check();
    void on_check();

public:
    Socket sock;

private:
    boost::signals2::signal<void(Connection)> close;

private:
    Endpoint remote;
    deadline_timer timer;
    boost::asio::streambuf read_buffer, write_buffer;
    boost::posix_time::ptime last_ping;
    std::string ip;
    std::size_t transfer_size;
}; // class TCPConnection

class ByteStormBoost
{
    static constexpr size_t kDefaultBufferSize{ 70 };
    static constexpr int kDefaultPort{ 8081 };

public:
    ByteStormBoost(int port, size_t buffer_size = kDefaultBufferSize);
    ~ByteStormBoost();

    void start();
    void stop();

    Connection create(size_t buffer_size, ProcessorBase<TCPConnection> *processor
= nullptr);
    void handle_accept(Connection &connection, const Error &err,
ProcessorBase<TCPConnection> *processor = nullptr);
    void on_delete_connection(Connection &connection);

public:
    io_service service;

    Acceptor *acc;
    Connections connections;

    ProcessorBase<TCPConnection> *processor{ nullptr };
```

```
    size_t buffer_size;
    int port;
}; // class ByteStormBoost

}; // namespace ByteStorm
```

Наследование:

- TCPConnection: Публично наследует от ByteStormBase и реализует интерфейс соединения для класса ByteStormBoost.
- TCPConnection: Является дружественным классом ByteStormBoost, что позволяет ByteStormBoost напрямую управлять экземплярами TCPConnection.

Данные-члены класса:

- transfer_size: Размер буфера передачи для соединения TCP.

Конструкторы:

- TCPConnection(service, transfer_size, h): Конструктор, который создает экземпляр TCPConnection, связывая его с сервисом ввода-вывода, устанавливая размер буфера передачи и назначая (необязательно) обработчик сообщений.
- ByteStormBoost(port, buffer_size): Конструктор, который создает экземпляр ByteStormBoost, назначая порт для прослушивания и размер буфера передачи.

Деструктор:

- ~TCPConnection(): Уничтожает экземпляр TCPConnection.
- ~ByteStormBoost(): Уничтожает экземпляр ByteStormBoost.

Функции-члены класса:

- flush(): Переопределяет метод ByteStormBase для сброса буфера записи.
- send(): Реализует отправку данных через сокет TCPConnection.
- read(): Инициирует чтение данных из сокета TCPConnection.

- read_complete(): Вызывается после завершения чтения данных.
- write(): Инициирует запись данных в сокет TCPConnection.
- write_complete(): Вызывается после завершения записи данных.
- on_connect(): Вызывается при установлении соединения TCPConnection.
- bind(): Функция шаблона, которая связывает функцию-член с объектом TCPConnection.
- check(): Проверяет соединение TCPConnection и закрывает его при необходимости.
- on_check(): Вызывается, когда функция check() завершает проверку.
- start(): Запускает сервис ввода-вывода и открывает порт для прослушивания.
- stop(): Останавливает сервис ввода-вывода и закрывает порт для прослушивания.
- create(): Создает и возвращает новый экземпляр TCPConnection.
- handle_accept(): Обрабатывает входящие соединения TCP и вызывает create() для их создания.
- on_delete_connection(): Удаляет соединение TCP из списка подключений.

Сигналы:

- TCPConnection::close: Сигнал, который вызывается при закрытии соединения TCPConnection.

Особенности:

- Класс ByteStormBoost обеспечивает сетевую функциональность для передачи и обработки сообщений на основе протокола TCP.
- Он использует библиотеку Boost.Asio для управления асинхронными сетевыми операциями.
- Класс TCPConnection представляет собой соединение TCP с буферами чтения и записи, а также с таймером проверки активности.
- ByteStormBoost управляет набором подключений TCPConnection и обрабатывает входящие соединения и события от подключенных клиентов.

Для корректного соединения сервера и клиента, а также их последующего взаимодействия было решено использовать шаблон проектирования - фабричный метод.

Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам (дочерним классам, субклассам) интерфейс

для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не конкретные классы, а манипулировать абстрактными объектами на более высоком уровне.

Класс **ByteStormBoost** является создателем объектов **TCPConnection**, который определяет интерфейс объектов, создаваемых абстрактным методом.

3.3. Модуль для работы с АЦП

```
using FSR = ADS1115::FullScaleRange;

enum Mode : uint8_t
{
    SingleShot,
    Loop
};

class ADCWorker : public WorkerBase
{
    static constexpr const char *kDefaultDevice{ "/dev/i2c-1" };
    static constexpr std::uint8_t kDefaultAddress{ 0x48 };
    static constexpr size_t kChannelCount{ 4 };
    static constexpr size_t kDefaultSWSPS{ 10 };
    static constexpr FSR kDefaultFSR{ FSR::FSR_0_256V };
    static constexpr bool kSilent{ true };

public:
    ADCWorker(RTDataBuffer *buffer = nullptr);
    ~ADCWorker();


    void handle(std::unique_ptr<Message::ClientData> &payload) override;
    void get(std::shared_ptr<std::uint8_t[]> &p) const override;
    void swSingleShot(size_t sps);

public:
    boost::signals2::signal<void(const float *mean)> signal;

private:
    size_t sw_sps{ kDefaultSWSPS };
    ADS1115::ADC<unix_i2c::i2c> adc;
    std::int16_t raw[kChannelCount];
    float mean_raw[kChannelCount];
    std::unordered_map<int, ADS1115::Multiplex> muxs;
    std::condition_variable cv;
```

```
    std::atomic_bool update{ false };

private:
    void prestart() override;
    ADS1115::Error readMux(ADS1115::Multiplex mux, std::int16_t &raw);
    ADS1115::Error readAll(std::int16_t *raw);

protected:
    void run() override;
};
```

Наследование:

- Является производным от класса WorkerBase, наследуя его функциональность обработки сообщений и управления потоками исполнения.

Данные-члены класса:

- sw_sps: Частота выборки, используемая для режима одиночного снимка (single shot).
- adc: Экземпляр класса ADS1115, представляющий аналого-цифровой преобразователь.
- raw: Массив из 4 целых чисел со знаком, хранящих необработанные значения АЦП для каждого канала.
- mean_raw: Массив из 4 чисел с плавающей запятой, хранящих средние значения необработанных значений АЦП для каждого канала.
- muxs: Неупорядоченная карта, сопоставляющая целочисленные идентификаторы с экземплярами мультиплексора ADS1115.
- cv: Условие, которое используется для синхронизации потоков.
- update: Атомарная булева переменная, указывающая, нужно ли обновлять данные.

Конструкторы:

- ADCWorker(buffer): Конструктор, который принимает указатель на буфер данных реального времени и создает экземпляр ADCWorker.

Деструктор:

- ~ADCWorker(): Уничтожает экземпляр ADCWorker.

Функции-члены класса:

- handle(payload): Переопределяет метод WorkerBase для обработки сообщений конфигурации, связанных с АЦП.
- get(p): Переопределяет метод WorkerBase для возврата данных, связанных с объектом ADCWorker.
- swSingleShot(sps): Устанавливает частоту выборки для режима одиночного снимка.
- readMux(mux, raw): Считывает необработанное значение АЦП для указанного мультиплексора.
- readAll(raw): Считывает необработанные значения АЦП для всех каналов.
- prestart(): Переопределяет метод WorkerBase для настройки устройства АЦП перед запуском потока исполнения.
- run(): Переопределяет метод WorkerBase для реализации поведения потока исполнения.

Сигналы:

- signal: Сигнал, который вызывается, когда вычисляются новые усредненные значения АЦП.

Особенности:

- Класс ADCWorker представляет собой специализированный класс, предназначенный для взаимодействия с аналого-цифровым преобразователем ADS1115.
- Он обеспечивает возможность считывать необработанные значения АЦП и вычислять их усреднение для нескольких каналов.
- Объект класса может работать как в режиме одиночного считывания, так и в продолжительном режиме, в зависимости от конфигурации.
- Использование сигналов и атомарных булевых переменных позволяет синхронизировать доступ к данным между потоками.

3.4. Код основной программы

Основной код программы необходим для корректной обработки пришедших байт с сетевого сокета каждым из модулей по отдельности. Было решено реализовать класс StandManager, отвечающий за это.

```
using namespace ByteStorm;

class StandManager : public Processor
{
```

```
    static constexpr size_t kDefaultWorkerCount{3};

public:
    StandManager(std::initializer_list<WorkerBase *> l);

    void append(WorkerBase *worker);
    void process(std::unique_ptr<uint8_t[]> &p, const size_t size) override;

    // slots
public:
    void onSingleShotCompleted(const float *mean);
    void onAngleSet(const Message::Instance &instance);
```

```
private:
    void processCommand(Commands::Common command);
    void sendErrorMessage(Message::Status err);

public:
    ByteStormBoost bs;

private:
    std::mutex interfaceMutex;
    std::vector<WorkerBase *> workers;
    Message::ServerPacket packet;
    RTDataBuffer buffer;
}; // class StandManager
```

Наследование:

- Является подклассом Processor, что позволяет ему обрабатывать сообщения в рамках фреймворка ByteStorm.

Данные-члены класса:

- workers: Вектор указателей на классы работников, представляющих различные компоненты стенда.
- packet: Объект ServerPacket, используемый для создания и отправки пакетов сообщений.
- buffer: Объект RTDataBuffer, используемый для хранения и обмена данными в реальном времени.

Конструкторы:

- StandManager(l): Конструктор, принимающий список указателей на классы работников и создающий экземпляр StandManager.

Функции-члены класса:

- append(worker): Добавляет указатель на класс работника в вектор workers.
- process(p, size): Переопределяет метод Processor для обработки входящих сообщений и их распределения соответствующим работникам.
- processCommand(command): Обрабатывает входящие команды и вызывает соответствующие методы классов работников.
- sendErrorMessage(err): Отправляет пакет сообщения об ошибке с указанным статусом.

Слоты:

- onSingleShotCompleted(mean): Вызывается, когда класс работника завершает режим одиночного снимка.
- onAngleSet(instance): Вызывается, когда класс работника получает сообщение об установке угла.

Особенности:

- Класс StandManager служит центральным компонентом, который управляет взаимодействием между различными классами работников в рамках стенда.
- Он отвечает за обработку входящих сообщений, их распределение и предоставление общего доступа к данным в реальном времени через объект RTDataBuffer.
- Использование слотов позволяет классу StandManager получать уведомления о событиях, происходящих в классах работников.

4. Заключение

В данной курсовой работе был описан процесс разработки программного обеспечения для управления поворотным стендом на базе микрокомпьютера с unix-подобной архитектурой ОС Raspberry Pi 4B. Были решены намеченные задачи и соблюдены поставленные критерии для его

работы. Также изучены принципы архитектуры многопоточных приложений и использования различных шаблонов проектирования. Основным результатом является корректная работа поворотного стенда в производственном плане.

5. Список литературы

1. John Torjo. Boost.Asio C++ Network Programming: Enhance Your Skills With Practical Examples for C++ Network Programming.
2. Кестер У. Аналого-цифровое преобразование

3. Федор Г. Пикус. Идиомы и паттерны проектрирования в современном C++.
4. Мейерс С. Эффективный и современный C++. 42 рекомендации по использованию C++11 и C++14.
5. А. В. Емельянов, А. Н. Шилин. Шаговые двигатели.
6. Уильям Стивенс. UNIX: Разработка сетевых приложений.