Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Системное программное обеспечение вычислительных машин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

РАЗРАБОТКА И РЕАЛИЗАЦИЯ ИМИТАЦИОННОЙ МОДЕЛИ

ИСТОЧНИКА НЕОДНОРОДНЫХ ТЕЛЕМЕТРИЧЕСКИХ ДАННЫХ

(СЕРВЕР)

БГУИР КП 6-05-0611-05 114 ПЗ

Студент: группы 350501, М. В. Козлов

Руководитель: старший препода- ватель каф. ЭВМ Поденок Л. П.

МИНСК 2025

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ

Заведующий кафедрой

\_\_\_\_\_\_\_\_ Б.В.Никульшин

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

ЗАДАНИЕ

по курсовому проектированию

Студенту Козлову Максиму Владимировичу

1. Тема проекта: Разработка и реализация имитационной модели источника неоднородных телеметрических данных (клиент)

2. Срок сдачи студентом законченного проекта: *15 мая 2025 г.*

3. Исходные данные к проекту: *язык программирования C*

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые

*Введение. 1. Обзор литературы. 2. Постановка задачи. 3. Системное проектирование. 4. Функциональное проектирование. 5. Разработка программных модулей. 6. Результат работы. Заключение. Список использованных источников. Приложения*

5. Перечень графического материала (с точным обозначением обязательных

чертежей и графиков):

1. *Схема структурная*
2. *Диаграмма классов*
3. *Ведомость документов*

6. Консультант по проекту *Поденок Л. П.*

7. Календарный график работы над проектом на весь период проектирования

(с обозначением сроков выполнения и трудоемкости отдельных этапов):

*раздел 1, 2 к 01.03 – 15 %;*

*разделы 3, 4 к 01.04 – 50 %;*

*разделы 5, 6, 7, 8, 9 к 01.05 – 80 %;*

*оформление пояснительной записки и графического материала к 15.05 – 100%*

*Защита курсового проекта с 29.05 по 09.06.*

РУКОВОДИТЕЛЬ  *Л. П. Поденок*

Задание принял к исполнению  *М. В. Козлов*

СОДЕРЖАНИЕ

[1 ОБЗОР ЛИТЕРАТУРЫ 7](#__RefHeading___Toc8822_1702547058)

[1.1 Обзор аналогов 7](#__RefHeading___Toc8824_1702547058)

[1.2 Постановка задачи 8](#__RefHeading___Toc8826_1702547058)

[2 ИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 10](#__RefHeading___Toc8828_1702547058)

[2.1 Модуль конфигурации и управления 10](#__RefHeading___Toc8830_1702547058)

[2.2 Модуль имитации данных 10](#__RefHeading___Toc8832_1702547058)

[2.3 Модуль сетевого взаимодействия 10](#__RefHeading___Toc8834_1702547058)

[2.4 Модуль сериализации и формирования протокола 11](#__RefHeading___Toc8836_1702547058)

[2.5 Модуль вспомогательных средств (утилиты) 11](#__RefHeading___Toc8838_1702547058)

[3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 12](#__RefHeading___Toc8840_1702547058)

[4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 12](#__RefHeading___Toc8842_1702547058)

[5 РЕЗУЛЬТАТ РАБОТЫ 12](#__RefHeading___Toc8844_1702547058)

[6 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 13](#__RefHeading___Toc8846_1702547058)

[7 ПРИЛОЖЕНИЕ А 14](#__RefHeading___Toc8848_1702547058)

[8 ПРИЛОЖЕНИЕ Б 15](#__RefHeading___Toc8850_1702547058)

[9 ПРИЛОЖЕНИЕ В 16](#__RefHeading___Toc8852_1702547058)

[10 ПРИЛОЖЕНИЕ Г 17](#__RefHeading___Toc8854_1702547058)

[11 ПРИЛОЖЕНИЕ Д 18](#__RefHeading___Toc8856_1702547058)

ВВЕДЕНИЕ

В современную эпоху цифровизации и повсеместного распространения IoT, систем промышленной автоматизации, аэрокосмических комплексов и систем мониторинга окружающей среды, непрерывный сбор и анализ данных от разнообразных датчиков и устройств стал критически важным. Эти данные, известные как телеметрия, предоставляют бесценную информацию о состоянии объектов, протекании процессов и параметрах окружающей среды в режиме реального времени. Эффективное управление, контроль и принятие обоснованных решений во многих отраслях напрямую зависят от своевременности и достоверности получаемых телеметрических данных.

Однако разработка, тестирование и отладка систем, предназначенных для сбора и обработки таких данных, сталкивается с существенными трудностями. Доступ к реальному оборудованию, генерирующему телеметрию, может быть ограничен, дорог или невозможен на ранних этапах разработки. Кроме того, реальные источники данных часто не позволяют гибко конфигурировать параметры генерируемых потоков для тестирования пограничных случаев, сценариев отказа или условий высокой нагрузки. Особенно остро эта проблема стоит при работе с **неоднородными** данными, когда одна система должна обрабатывать информацию разного типа (например, числовые показания температуры и давления, геолокационные координаты, текстовые статусные сообщения, бинарные данные изображений) и с разной частотой поступления от множества источников.

Для преодоления этих препятствий и обеспечения эффективного цикла разработки клиентских приложений необходимы инструменты, способные **имитировать** поведение реальных источников телеметрических данных. Создание реалистичной имитационной модели, способной генерировать потоки неоднородных данных, схожие по своим характеристикам (типам, частоте, диапазонам значений, динамике изменений) с реальными, становится ключевой задачей. Такая модель, реализованная в виде серверного приложения, может служить надежным и контролируемым источником данных для разработки и всестороннего тестирования клиентских систем сбора телеметрии.

**Целью данного проекта** является разработка и реализация имитационной модели источника неоднородных телеметрических данных в виде серверного приложения с использованием языка программирования C. Клиент должен быть способен принимать данные с одного или нескольких виртуальных устройств, принимать различные типы данных (числовые, строковые, структурированные, возможно, бинарные) с настраиваемой периодичностью и характеристиками, а также принимать эти данные по сети с использованием протокола TCP/IP для клиентского приложения сбора. Особое внимание будет уделено гибкости работы с этими данными для пользователя.

**Актуальность** данной работы обусловлена растущей потребностью в надежных и гибких инструментах для разработки и тестирования сложных систем сбора и обработки телеметрической информации в различных областях, от IoT и умных городов до промышленного мониторинга и научных исследований.

# ОБЗОР ЛИТЕРАТУРЫ

## Обзор аналогов

Для качественной оце­­нки разрабатываемого решения необходимо провести обзор аналогов. Важно отметить, что найти полный аналог, реализующий серверную имитацию­ неоднородных телеметрических данных на C++ с TCP-соединением для низколатентного сбора, затруднительно. Однако существуют классы инструментов, реализующие схожие концепции: предоставление серверной части для тестирования клиентских приложений и имитация источников данных, особенно в контексте тестирования API и разработки для IoT. Ниже рассмотрены некоторые из них.

В первую очередь стоит рассмотреть общие **Mock/Stub серверы**, предназначенные для тестирования API и сетевых клиентов. Ярким представителем является **MockServer** [1]. Это гибкий инструмент, позволяющий имитировать любые системы, с которыми код взаимодействует по протоколам HTTP/S. Он дает возможность детально настраивать ожидаемые запросы и соответствующие им ответы сервера, которые могут быть как статическими, так и динамически генерируемыми. Концептуально он схож с разрабатываемым проектом, так как предоставляет серверную часть для отладки и тестирования клиента. Ключевое отличие заключается в сильной ориентации MockServer (и аналогичных инструментов, таких как WireMock [2]) на веб-протоколы и часто на реализацию на Java или других языках, в то время как данный проект фокусируется на произвольном TCP-соединении, генерации специфичных телеметрических данных и реализации на C++.

Другим классом инструментов являются **симуляторы устройств и данных для платформ IoT**. Платформы вроде **AWS IoT Core [3]** или **Azure IoT Hub [4]** предлагают разработчикам SDK и примеры для создания виртуальных устройств, которые подключаются к облаку и отправляют потоки телеметрии (часто по MQTT или HTTPS), имитируя реальные датчики. Аналогично, open-source платформы, например, **ThingsBoard [5]**, могут включать компоненты для симуляции подключенных устройств. Эти инструменты полезны для тестирования облачной инфраструктуры или приложений, работающих с конкретной платформой. Их основная задача – генерация данных, схожих с телеметрическими – совпадает с целью проекта. Однако, они обычно функционируют как клиенты, привязанные к экосистеме своей платформы и её протоколам, в то время как разрабатываемый проект представляет собой автономный сервер, ожидающий подключения от кастомного клиента через TCP/IP и не зависящий от внешней инфраструктуры.

Наконец, часто для специфических задач или внутреннего тестирования разрабатываются **самописные скрипты-симуляторы** на языках вроде Python или Node.js. Они могут открывать TCP-порт и отправлять сгенерированные данные подключившемуся клиенту. Такие решения максимально гибки, но часто являются узкоспециализированными, могут не иметь удобных средств конфигурации и управления, а также могут уступать C++ реализации в производительности при генерации и отправке больших объемов данных.

Таким образом, хотя существуют инструменты для имитации серверов и генерации данных, они либо ориентированы на иные протоколы (HTTP, MQTT), либо являются частью крупных платформ, либо представляют собой менее структурированные решения. Разработка специализированного, автономного сервера на C++ для имитации источников неоднородных телеметрических данных с передачей по произвольному протоколу поверх TCP/IP остается актуальной задачей, особенно в контексте тестирования низколатентных систем сбора данных, что подтверждает практическую значимость разрабатываемого в данном проекте решения.

## Постановка задачи

После рассмотрения существующих аналогов можно заключить, что многие из них либо ориентированы на специфические протоколы (например, HTTP, MQTT) и платформы (AWS, Azure), либо представляют собой комплексные системы с широким функционалом, не всегда необходимым и реализуемым в рамках курсового проекта за отведенное время. В то же время, задача создания автономного, гибко настраиваемого сервера для имитации именно неоднородных телеметрических данных с передачей по чистому TCP/IP для тестирования специализированных клиентов остается актуальной.

Исходя из этого, были определены следующие ключевые задачи, которые будут решены в рамках данного курсового проекта:

* **Реализация серверного приложения:** Сервер должен быть способен прослушивать и принимать входящие TCP/IP соединения на заданном порту.
* **Имитация нескольких источников:** Сервер должен моделировать работу одного или нескольких виртуальных источников телеметрических данных.
* **Генерация неоднородных данных:** Должна быть реализована генерация данных различных предопределенных типов (например, числовые значения с плавающей точкой, целые числа, строки состояния, структурированные данные вроде GPS-координат).
* **Конфигурируемая генерация:** Необходимо предусмотреть возможность настройки параметров генерации для каждого источника и типа данных (например, частота отправки, диапазон значений, характер изменений).
* **Передача данных клиенту:** Сервер должен отправлять сгенерированные данные подключенным клиентам в соответствии с протоколом обмена.
* **Стабильная работа:** Сервер должен обеспечивать стабильную работу при подключении клиента и передаче данных в течение времени, необходимого для тестирования.

В качестве языка программирования выбран **C++**. Этот выбор обусловлен несколькими причинами: высокая производительность, которая может быть важна при генерации и отправке больших потоков данных; возможность низкоуровневой работы с сетевыми протоколами и представлением данных, что является ценным опытом в рамках курса "Операционные системы и сетевое программирование"; поддержка объектно-ориентированного подхода для логической организации имитационной модели (источники, типы данных, сервер); а также наличие у исполнителя опыта работы с данным языком.

Для реализации сетевого взаимодействия планируется использовать библиотеку **Boost.Asio**, предоставляющую мощные, кроссплатформенные и хорошо зарекомендовавшие себя абстракции для асинхронного (и синхронного) сетевого программирования на C++. Для эффективной упаковки (сериализации) неоднородных данных перед отправкой по сети будет рассмотрено применение библиотеки **Protocol Buffers**. Управление параметрами симуляции (например, количество источников, типы и частота данных) предполагается осуществлять через конфигурационные файлы, так как разработка полноценного графического интерфейса для серверного приложения не входит в основные задачи проекта. Сборка проекта будет управляться с помощью системы **CMake**.

Данный набор средств разработки и определенный перечень функциональных требований позволяют реализовать ключевые задачи проекта в установленные сроки и создать работоспособный сервер-симулятор телеметрических данных, необходимый для разработки и тестирования соответствующего низколатентного клиента сбора данных.

# СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После определения требований к функционалу разрабатываемого

приложения его следует разбить на функциональные блоки. Такой подход

упростит понимание проекта, позволит устранить проблемы в архитектуре,

обеспечит гибкость программного продукта в будущем путем добавления

новых блоков.

**2.1 Модуль конфигурации и управления**

Модуль конфигурации и управления необходим для задания начальных параметров работы клиента и координации запуска основных компонентов приложения. Его задача заключается в считывании настроек подключения (например, из файла или аргументов командной строки), таких как IP-адрес и сетевой порт сервера телеметрии. Также этот модуль отвечает за инициализацию других функциональных блоков (например, сетевого модуля, обработчика данных) и управление основным жизненным циклом клиентского приложения (запуск, установление соединения, цикл приема данных, завершение работы).

**2.2 Модуль сетевого взаимодействия**

Модуль сети предназначен для обеспечения связи клиента с сервернымприложением и необходим для приемателеметрических данных. Данный модуль должен отвечать за:

* Установление TCP-соединения с указанным сервером и портом.
* Прием потока байт от сервера.
* Управление состоянием активного соединения (обнаружение разрыва).
* Обработку сетевых ошибок при установлении соединения и приеме данных.
* Передачу полученных байтовых данных в модуль десериализации.

**2.3 Модуль десериализации и обработки протокола**

Модуль десериализации и обработки протокола необходим для преобразования последовательности байт, полученной по сети, во внутренние структуры данных приложения. Его задача – разобрать (десериализовать) входящий поток байт в соответствии с согласованным с сервером протоколом обмена, извлекая отдельные телеметрические сообщения (числа, строки, структуры). Он отвечает за:

* Буферизацию входящих данных для обработки сообщений, которые могут приходить частями.
* Идентификацию границ сообщений в потоке байт (например, по маркерам, префиксам длины).
* Десериализацию данных согласно формату протокола (включая обработку порядка байт – Endianness).
* Проверку целостности и корректности полученных сообщений (если протокол это предусматривает).
* Передачу восстановленных структур данных для дальнейшей обработки или отображения.

**2.4 Модуль представления и обработки данных**

Этот модуль отвечает за работу с полученными и десериализованными телеметрическими данными. В отличие от сервера, он не генерирует данные, а обрабатывает их. Функционал включает:

* Определение структур данных, используемых для представления телеметрии внутри клиента (эти определения должны совпадать с серверными).
* Реализацию логики обработки полученных данных:
  + Передачу данных другим системам или компонентам (если клиент является частью большей системы).
  + Выполнение каких-либо вычислений или проверок на основе полученных данных.

**2.5 Модуль вспомогательных средств (утилиты)**

Модуль вспомогательных средств необходим для предоставления общих функций, используемых другими модулями клиентского приложения. Сюда входят:

* Компоненты для логирования событий и ошибок.
* Вспомогательные функции для работы с сетью.
* Функции для работы с буферами данных.
* Утилиты для преобразования порядка байт.
* Вспомогательные функции для работы со временем и строками.

# ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описываются структуры, которые были

использованы при разработке утилиты.

**3.1 Структура**receive\_buffer\_t

Данная структура предназначена для управления буфером приема данных, поступающих от сервера по сетевому соединению. Она инкапсулирует динамически выделяемый участок памяти и отслеживает его заполненность, позволяя эффективно обрабатывать входящий поток байт, который может содержать неполные или несколько сообщений телеметрии. Структура receive\_buffer\_t включает в себя следующие поля:

* unsigned char \*buffer – указатель на динамически выделенную память, используемую в качестве буфера для хранения полученных байт.
* size\_t capacity – общая выделенная емкость буфера в байтах.
* size\_t length – текущее количество байт данных, хранящихся в буфере.

**3.2 Структура**telemetry\_data

Эта структура представляет собой основную единицу информации, получаемую от сервера – одно сообщение с телеметрическими данными от определенного источника. Она унифицирует представление различных типов данных (температура, давление, GPS координаты, статус). Структура telemetry\_data включает в себя следующие поля:

* int id – уникальный идентификатор источника телеметрических данных (например, датчика).
* telemetry\_data\_type type  – тип передаваемых данных, определяемый перечислением telemetry\_data\_type(например, DATA\_TYPE\_TEMPERATURE, DATA\_TYPE\_GPS и т.д.). Это поле определяет, какое из полей объединения valueсодержит актуальные данные.
* long long timestamp\_ms – временная метка получения данных на сервере (или их генерации), выраженная в миллисекундах с начала эпохи Unix.
* telemetry\_value value  – объединение (union), содержащее фактическое значение телеметрии. В зависимости от поля type, используется одно из следующих полей объединения:
  + float temperature  – значение температуры (используется, если type равен DATA\_TYPE\_TEMPERATURE).
  + float pressure  – значение давления (используется, если type равен DATA\_TYPE\_PRESSURE).
  + float humidity  – значение влажности (используется, если type равен DATA\_TYPE\_HUMIDITY).
  + gps\_data gps – структура (struct gps\_data), содержащая координаты GPS:
    - double latitude  – широта.
    - double longitude – долгота.  
      (используется, если type равен DATA\_TYPE\_GPS).
  + char status[20] – строковое представление статуса устройства (используется, если type равен DATA\_TYPE\_STATUS).

# РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе рассматриваются алгоритмы работы ключевых функций и логических блоков клиентского приложения, отвечающих за установку соединения, прием данных от сервера, их буферизацию, разбор (десериализацию) и отображение. Основное внимание уделено логике основного цикла обработки данных в функции main и функции десериализации deserialize\_telemetry\_data, которая преобразует поток байт в структурированные телеметрические сообщения.

**4.1 Разработка схем алгоритмов**

Для наглядного представления логики работы основных компонентов программы были разработаны схемы алгоритмов. *.*

* Схема алгоритма основного цикла приема и обработки данных (реализованного в функции main) приведена в приложении А.
* Схема алгоритма функции десериализации телеметрических данных deserialize\_telemetry\_data приведена в приложении Б.

**4.2 Разработка алгоритмов**

Ниже приведено пошаговое описание алгоритмов для выбранных программных модулей.

**Алгоритм основного цикла приема и обработки данных (в функции**main**)**:

1. Начало бесконечного цикла (while(1)).
2. **Проверка и подготовка буфера:**Вызов функции ensure\_buffer\_capacity() для структуры recv\_buf, чтобы убедиться, что в буфере есть достаточно места (например, RECV\_BUFFER\_SIZE байт) для приема новой порции данных.
3. Если ensure\_buffer\_capacity() вернула ошибку (например, не удалось выделить память), вывести сообщение об ошибке и перейти к шагу 16 (завершение цикла).
4. **Прием данных:** Вызов функции recv() для чтения данных из сокета sockfd в свободное пространство буфера recv\_buf(начиная с recv\_buf.buffer + recv\_buf.length). Сохранить количество принятых байт в переменную bytes\_received.
5. **Анализ результата recv():**
   * Если bytes\_received < 0: Проверить errno. Если errno равно EINTR (вызов прерван сигналом), вернуться к шагу 4 (повторить чтение). Иначе, считать ошибкой чтения, вывести сообщение об ошибке (perror) и перейти к шагу 16.
   * Если bytes\_received == 0: Сервер закрыл соединение. Вывести информационное сообщение и перейти к шагу 16.
   * Если bytes\_received > 0: Данные успешно получены. Увеличить recv\_buf.length на bytes\_received. Перейти к шагу 6.
6. **Обработка накопленных данных:** Инициализировать счетчик обработанных байт за итерацию processed\_in\_cycle = 0.
7. Начало внутреннего цикла (while (recv\_buf.length > processed\_in\_cycle)), пока в буфере есть необработанные данные.
8. Определить указатель на начало необработанных данных (data\_start = recv\_buf.buffer + processed\_in\_cycle) и их доступную длину (data\_len = recv\_buf.length - processed\_in\_cycle).
9. Вызвать функцию deserialize\_telemetry\_data(data\_start, data\_len, &current\_data) для попытки разобрать одно телеметрическое сообщение из начала необработанных данных. Сохранить результат (количество обработанных байт или код ошибки/состояния) в processed\_now.
10. **Анализ результата**deserialize\_telemetry\_data()**:**
    * Если processed\_now > 0: Сообщение успешно разобрано. Вызвать функцию print\_telemetry\_data(&current\_data)для вывода данных. Увеличить processed\_in\_cycle на processed\_now. Вернуться к шагу 7 (продолжить обработку оставшихся данных в буфере).
    * Если processed\_now == 0: Данных в буфере недостаточно для полного сообщения. Прервать внутренний цикл (break) и перейти к шагу 13 (ожидать поступления новых данных).
    * Если processed\_now < 0 (ошибка) или processed\_now == 1 (пропущен байт маркера): Произошла ошибка при разборе или обнаружен некорректный байт. Вывести сообщение об ошибке/пропуске. Увеличить processed\_in\_cycle на 1 (для сдвига мимо проблемного байта). Вернуться к шагу 7.
11. Конец внутреннего цикла обработки (из шага 7).
12. **Сдвиг буфера:** Если processed\_in\_cycle > 0 (были обработаны какие-то данные):
13. Если processed\_in\_cycle < recv\_buf.length (в буфере остался неполный пакет или необработанные данные), переместить оставшиеся recv\_buf.length - processed\_in\_cycle байт в начало буфера с помощью memmove().
14. Уменьшить recv\_buf.length на processed\_in\_cycle.
15. Вернуться к шагу 1 (начало основного цикла для ожидания новых данных).
16. Конец основного цикла (выход из while(1)).
17. Конец алгоритма (далее следует закрытие сокета и освобождение буфера).

**Функция deserialize\_telemetry\_data():**

1. Начало.
2. **Входные данные:**
   * const unsigned char \*buffer – указатель на буфер, содержащий последовательность байт для десериализации.
   * size\_t buffer\_len – количество байт, доступных в буфере.
   * telemetry\_data \*data – указатель на структуру, куда будут записаны десериализованные данные.
3. **Выходные данные:** ssize\_t – результат десериализации:
   * > 0: Количество успешно обработанных байт, составляющих одно полное сообщение.
   * 0: Недостаточно данных в буфере для полного сообщения.
   * -1: Неустранимая ошибка формата данных (например, неверный тип).
   * 1: Обнаружен некорректный байт-маркер; этот байт следует пропустить.
4. Объявление переменных: size\_t offset (текущее смещение в буфере), временные переменные для хранения сетевых представлений чисел (net\_id, type\_byte, net\_timestamp, net\_float, net\_lat, net\_lon), size\_t required\_payload\_size (ожидаемый размер полезной нагрузки).
5. Проверка входных указателей buffer и data на NULL. Если хотя бы один NULL, вернуть -1.
6. Проверка buffer\_len на 0. Если 0, вернуть 0.
7. Инициализировать offset = 0.
8. **Проверка маркера:** Проверить значение buffer[offset].
9. Если buffer[offset] не равно 'T', вывести сообщение об ошибке, вернуть 1 (сигнал пропустить этот байт).
10. Увеличить offset на sizeof(char).
11. **Проверка длины заголовка:** Проверить, достаточно ли оставшихся данных (buffer\_len - offset) для чтения минимального заголовка (ID, тип, временная метка). Если недостаточно, вернуть 0.
12. **Чтение ID:** Скопировать sizeof(uint32\_t) байт из buffer + offset во временную переменную net\_id. Преобразовать net\_id из сетевого порядка байт в хостовый (ntohl()) и сохранить результат в data->id. Увеличить offset на sizeof(uint32\_t).
13. **Чтение типа:** Скопировать sizeof(uint8\_t) байт из buffer + offset в type\_byte. Проверить, является ли type\_byte допустимым значением перечисления telemetry\_data\_type. Если недопустимый, вывести ошибку и вернуть -1. Преобразовать type\_byte к типу telemetry\_data\_type и сохранить в data->type. Увеличить offset на sizeof(uint8\_t).
14. **Чтение временной метки:** Скопировать sizeof(uint64\_t) байт из buffer + offset в net\_timestamp. Преобразовать net\_timestamp из сетевого порядка байт в хостовый (ntohll()) и сохранить результат в data->timestamp\_ms. Увеличить offset на sizeof(uint64\_t).
15. **Определение размера полезной нагрузки:** Используя switch по data->type, определить ожидаемый размер полезной нагрузки (required\_payload\_size) для данного типа сообщения (sizeof(uint32\_t) для float, 2 \* sizeof(uint64\_t) для GPS, sizeof(data->value.status) для статуса).
16. **Проверка длины полезной нагрузки:** Проверить, достаточно ли оставшихся данных (buffer\_len - offset) для чтения required\_payload\_size. Если недостаточно, вернуть 0.
17. **Чтение значения (полезной нагрузки):** Используя switch по data->type:
    * **Температура/Давление/Влажность:** Скопировать sizeof(uint32\_t) байт в net\_float. Преобразовать net\_float в float (ntohf()) и сохранить в соответствующее поле data->value. Увеличить offset.
    * **GPS:** Скопировать sizeof(uint64\_t) в net\_lat, затем еще sizeof(uint64\_t) в net\_lon. Преобразовать каждое значение в double (ntohd()) и сохранить в data->value.gps.latitude и data->value.gps.longitude. Увеличить offset на 2 \* sizeof(uint64\_t).
    * **Статус:** Скопировать sizeof(data->value.status) байт напрямую в data->value.status. Убедиться, что строка завершается нулевым символом (на всякий случай). Увеличить offset на sizeof(data->value.status).
18. Вернуть итоговое значение offset, которое теперь равно общему количеству байт, успешно прочитанных для этого сообщения.
19. Конец.

# РЕЗУЛЬТАТ РАБОТЫ

Функционал программы несложно проверить с помощью флеш-

накопителя небольшой емкости или любого файла небольшого размера,

чтобы программа не читала информацию слишком долго.

Если пользователь количество параметров командной строки менее

трех, он увидит на экране окно помощи, в котором дан шаблон для запуска

программы и описание возможных опций.

Для дальнейшей работы нужно смонтировать диск. Для этого требуется

создать отдельный раздел. Пример: sudo mkdir /media/darya/ISO.

Для последующего монтирования диска нужно узнать название

спасаемого устройство и его файловую систему. Это можно осуществить с

помощью команды fdisk –l. Для тестового устройства файловой системой

является FAT32.

Для проверки программы производится запуск программы со

следующей командной строкой: sudo ./myrescue /dev/sda1 myres.

Dev/sda1 — это имя флеш-накопителя, с помощью которого осуществлялось

тестирование, myres — имя выходного файла. Настройки соответствуют

чтению от начала до конца.

Терминал во время выполнения программы:

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы были углублены знания о языке

программирования Си, получены и закреплены знания об устройстве POSIX-

совместимых файловых систем, работе с файлом при помощи файлового

дескриптора, сигналах и другие знания в области системного

программирования.

В ходе выполнения курсовой работы была реализована утилита

восстановления плохих секторов путем многократного чтения.

Тем не менее, программа определённо имеет потенциал к улучшению,

так как данный курсовой проект рассчитан на студентов второго курса.

Данный курсовой проект дал мне ценный опыт в области написания

самодостаточных программ в целом и в POSIX-системах в частности. Следует

тщательно продумать весь путь создания проекта сразу, предварительно

изучив теоретический материал по техническому заданию.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] **MockServer** [Электронный ресурс]. – **MockServer.** – Режим доступа: https://www.mock-server.com/ – Дата доступа: 10.03.2025

[2] WireMock [Электронный ресурс]. – WireMock. – Режим доступа: https://wiremock.org/ – Дата доступа: 10.03.2025

[3] A**mazon** [Электронный ресурс]. – **AWS IoT Core** – Режим доступа: https://aws.amazon.com/ru/iot-core/ – Дата доступа: 10.03.2025

[4] M**icrosoft** [Электронный ресурс]. – **Azure IoT Hub** – Режим доступа: https://azure.microsoft.com/en-us/products/iot-hub – Дата доступа: 10.03.2025

[5] E**ngix-ltd** [Электронный ресурс]. – **ThingsBoard**. – Режим доступа: https://engix-ltd.github.io/ – Дата доступа: 10.03.2025

# ПРИЛОЖЕНИЕ А

(Обязательное)

**Ведомость документов**

# ПРИЛОЖЕНИЕ Б

(Обязательное)

**Структурная схема**

# ПРИЛОЖЕНИЕ В

(Обязательное)

**Диаграмма классов**

# ПРИЛОЖЕНИЕ Г

(Обязательное)

**Схема программы**

# ПРИЛОЖЕНИЕ Д

(Обязательное)

**Код программы**