**СОДЕРЖАНИЕ**

[Введение 5](#_Toc197344565)

[1 Обзор литературы 6](#_Toc197344566)

[1.1 Утилита обнаружения и тестирования функций USB-устройства как проект 6](#_Toc197344567)

[1.2 Анализ существующих аналогов 6](#_Toc197344568)

[1.3 Постановка задачи 7](#_Toc197344569)

[2 Системное проектирование 7](#_Toc197344570)

[2.1 Модуль запуска и управления (Application + main) 8](#_Toc197344571)

[2.2 Модуль мониторинга и идентификации устройств (UdevMonitor + часть Application) 8](#_Toc197344572)

[2.3 Модуль тестирования (DeviceTester) 8](#_Toc197344573)

[2.4 Модуль отображения результатов (ResultDisplay) 9](#_Toc197344574)

[2.5 Модуль служебного логгирования (Интегрирован) 9](#_Toc197344575)

[3 Функциональное проектирование 10](#_Toc197344576)

[3.1 Разработка диаграммы классов 10](#_Toc197344577)

[3.2 Описание классов 10](#_Toc197344578)

[4 Разработка программных модулей 14](#_Toc197344579)

[4.1 Разработка схем алгоритмов 14](#_Toc197344580)

[4.2 Разработка алгоритмов 14](#_Toc197344581)

[5 Результаты работы 15](#_Toc197344582)

Введение

В современном мире USB-устройства играют ключевую роль в работе компьютеров и других электронных систем, обеспечивая подключение периферийных устройств, передачу данных и взаимодействие с различными аппаратными компонентами. Однако процесс обнаружения и диагностики USB-устройств часто требует специализированных инструментов, особенно при работе на низком уровне, где необходимо прямое взаимодействие с системными API и драйверами.

Целью данного курсового проекта является разработка утилиты на языке C++ для обнаружения и тестирования USB-устройств в операционной системе Linux. Программа должна предоставлять пользователю детальную информацию о подключённых устройствах, включая идентификаторы производителя, модель, скорость передачи данных и другие параметры, а также выполнять базовые тесты на работоспособность, такие как проверка скорости чтения, записи и целостности передаваемой информации.

Актуальность проекта заключается в необходимости создания удобного и эффективного инструмента для диагностики USB-устройств, который может быть полезен как для системных администраторов, так и для обычных пользователей. В процессе разработки будут использованы низкоуровневые библиотеки, такие как libusb, а также системные API операционных систем для обеспечения максимальной точности и контроля над процессами взаимодействия с устройствами.

Основные функции утилиты включают:

— Перечисление всех подключённых USB-устройств с выводом технических характеристик;

— Тестирование скорости передачи данных между устройством и компьютером;

— Проверку целостности данных при чтении и записи;

— Сохранение результатов тестирования в лог-файл для дальнейшего анализа.

Проект имеет практическую ценность, так как подобные инструменты востребованы при диагностике неисправностей, а также при оценке производительности USB-устройств. В перспективе программа может быть расширена за счёт добавления функций автоматического мониторинга и интеграции с другими системными утилитами.

1 Обзор литературы

1.1 Утилита обнаружения и тестирования функций USB-устройства как проект

Разработка утилиты для обнаружения и тестирования USB-устройств представляет собой важный практический проект в области системного программирования. Такой инструмент позволяет не только получать базовую информацию о подключённых устройствах, но и проводить комплексную диагностику их работоспособности. В процессе создания утилиты особое внимание уделяется взаимодействию с низкоуровневыми API операционных систем и библиотеками, такими как libusb, что обеспечивает прямой доступ к функционалу USB-контроллеров и подключённых устройств.

Актуальность проекта обусловлена необходимостью удобного и универсального инструмента для диагностики USB-устройств, который мог бы использоваться как в повседневной работе, так и при решении специализированных задач. Создаваемая утилита на языке C++ сможет не только отображать технические характеристики устройств, но и тестировать их производительность, что делает её полезной для системных администраторов, разработчиков и обычных пользователей. Реализация подобного проекта позволяет глубже понять принципы работы USB-стека в современных операционных системах и получить ценный опыт в области системного программирования.

1.2 Анализ существующих аналогов

Прежде чем приступить к разработке собственного решения, необходимо изучить существующие аналогичные утилиты для работы с USB-устройствами. Анализ функциональных возможностей, преимуществ и недостатков подобных инструментов позволит определить оптимальный подход к реализации проекта, а также выделить уникальные особенности разрабатываемой программы.

1.2.1 USBView

USBView – стандартная утилита, входящая в состав Windows SDK, которая предоставляет детальную информацию о подключённых USB-устройствах и их характеристиках. Программа отображает иерархию USB-портов, идентификаторы устройств (Vendor ID, Product ID), а также поддерживаемые скорости передачи данных. Однако USBView имеет ограниченный функционал в части тестирования производительности и не предоставляет инструментов для проверки стабильности работы устройств.

1.2.2 Isusb

Утилита lsusb, доступная в Linux, позволяет выводить список подключённых USB-устройств с их основными параметрами. Она поддерживает фильтрацию по классам устройств и вывод дополнительных дескрипторов. Несмотря на свою простоту и удобство использования в терминале, lsusb не предоставляет возможностей для тестирования скорости передачи данных или диагностики ошибок, что ограничивает её применение в задачах глубокого анализа.

1.2.3 USBDeview

USBDeview — программа от NirSoft, которая сочетает в себе функции мониторинга и управления USB-устройствами. Она позволяет не только просматривать подключённые устройства, но и отключать их, сохранять историю подключений, а также экспортировать данные в различные форматы. Однако, как и предыдущие аналоги, USBDeview не включает инструментов для тестирования производительности и проверки целостности данных, что делает её менее пригодной для задач комплексной диагностики.

1.3 Постановка задачи

Разработка утилиты для обнаружения и тестирования USB-устройств на платформе Linux ставит ряд технических задач, требующих решения средствами языка C++ и низкоуровневых Linux-интерфейсов. Основная цель проекта — создать консольное приложение, способное не только идентифицировать подключенные USB-устройства через системные механизмы ядра Linux, но и проводить комплексную проверку их функциональности и производительности.

Ключевыми задачами являются: реализация парсинга данных из подсистемы udev для получения полной информации об устройствах; разработка модуля тестирования с использованием библиотеки libusb для проверки скорости передачи данных и целостности соединения; создание системы логирования результатов в машиночитаемом формате. Особое внимание уделяется оптимизации работы с raw-устройствами через интерфейсы HID и массовой памяти, а также обработке возможных ошибок ввода-вывода.

Проект разрабатывается исключительно для Linux с использованием специфичных для этой ОС технологий: udev для мониторинга устройств и др. Результатом станет специализированный инструмент, превосходящий по функциональности стандартные утилиты в части тестирования производительности USB-устройств под Linux.

2 Системное проектирование

Системное проектирование данной утилиты направлено на создание фонового сервиса для операционной системы Linux, способного автоматически обнаруживать подключаемые USB-устройства, идентифицировать их тип, проводить тестирование для накопителей информации и отображать сводную информацию пользователю. Основной акцент сделан на событийной модели обработки подключений и разделении логики на функциональные модули.

2.1 Модуль запуска и управления (Application + main)

Отвечает за инициализацию приложения, парсинг аргумента командной строки (-d) для определения режима работы (foreground или демон), запуск/остановку основного цикла и управление жизненным циклом приложения. Включает в себя:

— Обработку сигналов операционной системы (SIGINT, SIGTERM) для корректного завершения.

— Вызов функции демонизации при необходимости.

— Инициализацию и координацию работы других модулей.

2.2 Модуль мониторинга и идентификации устройств (UdevMonitor + часть Application)

Ключевой модуль, отвечающий за взаимодействие с системой управления устройствами Linux:

— Использует библиотеку libudev для асинхронного мониторинга событий подключения (add) и отключения (remove) устройств.

— Фильтрует события для подсистем usb (тип usb\_device) и block.

— При обнаружении события usb add:

— Извлекает базовую информацию об устройстве (VID, PID, производитель, имя) через sysfs.

— Определяет, вероятно ли устройство является накопителем, проверяя класс его USB-интерфейса (Mass Storage Class 0x08).

— Сохраняет информацию об устройстве для последующей обработки.

— При обнаружении события block add:

— Проверяет свойства блочного устройства (ID\_BUS=usb, ID\_TYPE=disk, DEVTYPE=disk), чтобы подтвердить, что это USB-накопитель.

— Связывает блочное устройство с ранее обнаруженным родительским USB-устройством.

— Передает информацию об обработанных событиях управляющему модулю (Application).

2.3 Модуль тестирования (DeviceTester)

Выполняет тестирование производительности и целостности данных только для устройств, идентифицированных как USB-накопители:

— Осуществляет прямое чтение/запись на блочное устройство (например, /dev/sdX) с использованием системных вызовов open, read, write, lseek.

— Проводит тест скорости записи и чтения путем передачи определенного объема данных.

— Выполняет проверку целостности записанных данных путем их последующего чтения и сравнения.

— Формирует текстовый отчет о результатах тестов (скорость, успех/неудача целостности).

2.4 Модуль отображения результатов (ResultDisplay)

Отвечает за подготовку и отображение сводной информации и результатов тестирования пользователю:

— Создает временный файл (в /var/tmp) для записи отчета.

— Записывает в файл основную информацию об обнаруженном USB-устройстве.

— Вызывает Модуль тестирования (если устройство является накопителем) и дописывает результаты тестов в тот же файл.

— Пытается запустить внешнюю утилиту zenity для отображения содержимого временного файла в графическом окне.

— Удаляет временный файл после попытки отображения.

2.5 Модуль служебного логгирования (Интегрирован)

Реализует журналирование внутренних событий программы для отладки и мониторинга:

— Использует стандартный механизм syslog Linux.

— Записывает события старта/остановки, обнаружения устройств, этапы обработки, ошибки инициализации, результаты тестов и попытки вызова zenity.

— Использует разные идентификаторы (usb\_monitor\_fg или usb\_monitor) в зависимости от режима запуска.

Структурная схема проекта представлена в приложении А.

3 Функциональное проектирование

В данном разделе описываются входные и выходные данные программы, диаграмма классов, а также приводится описание используемых классов и их методов.

3.1 Разработка диаграммы классов

Диаграмма классов представлена в приложении Б.

3.2 Описание классов

Структура DeviceInfo — простой контейнер данных, предназначенный для хранения собранной информации о конкретном USB-устройстве, обнаруженном в системе. Не имеет базовых классов или сложной логики.

Структура содержит следующие поля:

— std::string devpath — уникальный путь устройства в sysfs (например, /devices/pci.../usbX/X-X);

— std::string vendor\_id — идентификатор производителя (VID);

— std::string product\_id — идентификатор продукта (PID);

— std::string manufacturer — строка с именем производителя (может быть пустой);

— std::string product\_name — строка с именем устройства (может быть пустой);

— std::string block\_device — путь к соответствующему блочному устройству (например, /dev/sda), если оно найдено и идентифицировано;

— bool results\_displayed — флаг, указывающий, было ли уже показано окно с информацией/результатами для этого USB-устройства;

— bool is\_likely\_storage — флаг, указывающий, определило ли приложение по USB-интерфейсам, что это устройство вероятно является накопителем (требует дальнейшей проверки через событие block).

Пространство имен DaemonUtil предоставляет утилитарную функциональность для перевода процесса в режим фонового демона в Linux. Использует стандартные POSIX-функции (fork, setsid, signal, chdir, umask, close, open, dup2) и syslog для логирования ошибок во время демонизации.

Пространство имен содержит следующую функцию:

— void daemonize() — выполняет стандартную последовательность действий для отсоединения процесса от управляющего терминала и перевода его в фон.

Класс DeviceTester инкапсулирует логику проведения тестов производительности и целостности данных для USB-накопителей. Все методы класса статические, так как не требуется хранить состояние между вызовами тестов для разных устройств. Использует низкоуровневые файловые операции POSIX (open, read, write, lseek, close) для прямого доступа к блочным устройствам и syslog для логирования процесса тестирования.

Класс содержит следующие методы:

— static void performTests(const std::string& block\_dev\_path, std::ostream& out\_stream) — публичный статический метод, выполняющий тесты записи, чтения и проверки целостности для указанного блочного устройства и записывающий подробные результаты в переданный поток out\_stream;

— private static long long current\_time\_ms() — вспомогательный метод для получения текущего времени в миллисекундах;

— private static bool check\_data\_integrity(const char\* buffer\_written, const char\* buffer\_read, size\_t size)— вспомогательный метод для сравнения буферов при проверке целостности данных.

Класс ResultDisplay отвечает за конечную обработку информации об устройстве: запись в файл, вызов тестов (через DeviceTester) и попытку отображения результатов пользователю через внешнюю утилиту zenity. Метод класса статический. Использует стандартные файловые операции C (mkstemp, fdopen, fprintf, fclose, unlink) для работы с временным файлом в /var/tmp, syslog для логирования и system() для запуска zenity.

Класс содержит следующие методы:

— static void prepareAndDisplay(const DeviceInfo& info, bool is\_storage\_device) — публичный статический метод, который создает временный файл, записывает в него базовую информацию об устройстве info, вызывает DeviceTester::performTests, если is\_storage\_device истинно, дописывает результаты тестов, пытается запустить zenity для показа файла и затем удаляет временный файл.

Класс UdevMonitor инкапсулирует взаимодействие с библиотекой libudev для мониторинга событий подключения/отключения устройств. Отвечает за инициализацию libudev, настройку фильтров событий и запуск основного цикла ожидания событий с помощью select. Использует syslog для логирования своей работы.

Класс определяет следующие типы:

— using DeviceEventCallback = std::function<void(struct udev\_device\* dev)> — тип функции обратного вызова, которая будет вызвана при получении события udev.

Класс содержит следующие поля:

— private struct udev\* udev\_context\_ — указатель на контекст libudev;

— private struct udev\_monitor\* udev\_monitor\_ — указатель на монитор событий udev;

— private int udev\_fd\_ — файловый дескриптор монитора для select;

— private std::atomic<bool>& running\_flag\_ — ссылка на внешний атомарный флаг, управляющий работой цикла мониторинга (принадлежит классу Application).

Класс содержит следующие методы:

— public UdevMonitor(std::atomic<bool>& running\_flag) — конструктор, принимающий ссылку на флаг управления циклом;

— public ~UdevMonitor() — деструктор, освобождающий ресурсы libudev;

— public bool initialize() — инициализирует libudev, создает монитор, устанавливает фильтры на события usb/usb\_device и block, включает получение событий и получает файловый дескриптор;

— public void run(DeviceEventCallback callback) — запускает основной цикл, который ожидает событий на udev\_fd\_ с помощью select, при получении события вызывает udev\_monitor\_receive\_device и передает полученное устройство udev\_device в функцию callback. Цикл продолжается, пока running\_flag\_ истинна;

— public void stop() — устанавливает running\_flag\_ в false, сигнализируя циклу run о необходимости завершения.

Класс Application является главным управляющим классом приложения. Он координирует работу всех остальных модулей, инициализирует систему, обрабатывает сигналы ОС и реализует основную логику обработки событий udev, полученных от UdevMonitor. Использует syslog для логгирования своей работы и хранит состояние обнаруженных устройств.

Класс содержит следующие поля:

— private bool is\_daemon\_ — флаг, указывающий, работает ли приложение в режиме демона;

— private UdevMonitor udev\_monitor\_ — экземпляр класса монитора udev;

— private std::map<std::string, DeviceInfo> active\_devices\_map\_ — карта для хранения информации об активных USB-устройствах (ключ — devpath USB-устройства);

— private std::atomic<Application\*> instance\_ — указатель на единственный экземпляр Application для доступа из обработчика сигналов;

— private static std::atomic<bool> running\_ — статический атомарный флаг, управляющий основным циклом работы приложения и UdevMonitor.

Класс содержит следующие методы:

— public Application(bool as\_daemon) — конструктор, инициализирует флаг is\_daemon\_, syslog и вызывает DaemonUtil::daemonize, если необходимо;

— public ~Application() — деструктор, вызывает cleanup;

— public int run() — основной метод, запускающий приложение: вызывает initialize, затем udev\_monitor\_.run() с передачей onDeviceEvent в качестве callback, и после завершения цикла вызывает cleanup;

— private bool initialize() — инициализирует udev\_monitor\_ и устанавливает обработчики сигналов;

— private void cleanup() — закрывает syslog;

— private void setupSignalHandlers() — устанавливает статический обработчик для SIGINT и SIGTERM;

— private static void staticSignalHandler(int signum) — статический метод, вызываемый ОС при получении сигнала; находит экземпляр Application и вызывает его метод handleSignal;

— private void handleSignal(int signum) — устанавливает флаг running\_ в false и вызывает udev\_monitor\_.stop();

— private void onDeviceEvent(struct udev\_device\* dev) — метод обратного вызова, передаваемый в UdevMonitor::run. Анализирует тип события (usb add, block add, usb remove), вызывает hasMassStorageInterface для usb add, сохраняет/обновляет информацию в active\_devices\_map\_ и вызывает ResultDisplay::prepareAndDisplay в нужный момент (сразу для не-накопителей, после block add для накопителей), управляя флагом results\_displayed;

— private bool hasMassStorageInterface(struct udev\_device\* usb\_dev) — вспомогательный метод для проверки наличия интерфейса класса Mass Storage (0x08) у USB-устройства.

4 Разработка программных модулей

В данном разделе представлены схемы алгоритмов и алгоритмы по шагам основных функций, разработанных в рамках курсового проекта.

4.1 Разработка схем алгоритмов

Схема алгоритма DaemonUtil::daemonize представлена в приложении В, схема алгоритма ResultDisplay::prepareAndDisplay представлена в приложении Г.

4.2 Разработка алгоритмов

4.2.1 Алгоритм по шагам функции Application::initialize

Шаг 1. Записать в syslog отладочное сообщение о начале инициализации приложения.

Шаг 2. Вызвать метод initialize() для объекта udev\_monitor\_.

Шаг 3. Если результат вызова udev\_monitor\_.initialize() равен false:

Шаг 4. Записать в syslog критическую ошибку инициализации UdevMonitor.

Шаг 5. Вернуть false.

Шаг 6. Вызвать метод setupSignalHandlers() для установки обработчиков сигналов ОС.

Шаг 7. Записать в syslog отладочное сообщение об успешном завершении инициализации приложения.

Шаг 8. Вернуть true.

4.2.2 Алгоритм по шагам функции UdevMonitor::initialize

Шаг 1. Создать новый контекст udev с помощью udev\_new() и сохранить указатель в udev\_context\_.

Шаг 2. Если udev\_context\_ равен nullptr (ошибка создания), записать ошибку в syslog и вернуть false.

Шаг 3. Создать новый монитор udev из netlink (udev\_monitor\_new\_from\_netlink) и сохранить указатель в udev\_monitor\_.

Шаг 4. Если udev\_monitor\_ равен nullptr (ошибка создания), записать ошибку в syslog и вернуть false.

Шаг 5. Добавить к монитору фильтр для отслеживания событий подсистемы usb с типом usb\_device.

Шаг 6. Добавить к монитору фильтр для отслеживания событий подсистемы block (любого типа).

Шаг 7. Включить получение событий для монитора с помощью udev\_monitor\_enable\_receiving().

Шаг 8. Если включение не удалось (вернулось < 0), записать ошибку в syslog и вернуть false.

Шаг 9. Получить файловый дескриптор монитора с помощью udev\_monitor\_get\_fd() и сохранить его в udev\_fd\_.

Шаг 10. Если дескриптор некорректен (udev\_fd\_ < 0), записать ошибку в syslog и вернуть false.

Шаг 11. Записать в syslog сообщение об успешной настройке монитора.

Шаг 12. Вернуть true.

5 Результаты работы

