



Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»
КАФЕДРА ИУ7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ
НА ТЕМУ:
«Драйвер для отображения бегущего текста на LCD
дисплее с использованием Raspberry Pi»

Студент **ИУ7-76Б**

_____ **Е. А. Куликов**

(Подпись, дата)

(И.О.Фамилия)

Руководитель

_____ **Н.Ю. Рязанова**

(Подпись, дата)

(И.О.Фамилия)

Рекомендованная руководителем Кур оценка: _____

2025 г.

**Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)
(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ
Заведующий кафедрой ИУ7
_____ И. В. Рудаков
«__» сентября 2025 г.

**З А Д А Н И Е
на выполнение курсовой работы**

по теме

*Драйвер для отображения бегущего текста на LCD дисплее
с использованием Raspberry Pi*

Студент группы ИУ7-76Б

Куликов Егор Андреевич

Направленность КуР (учебная, исследовательская, практическая, производственная, др.):
учебная.

Источник тематики (кафедра, предприятие, НИР): кафедра.

График выполнения КуР: 25% к 5 нед., 50% к 8 нед., 75% к 11 нед., 100% к 15 нед.

Техническое задание

Разработать драйвер для одноплатного компьютера Raspberry Pi для отображения вводимого пользователем текста на LCD дисплее в формате статичной или бегущей строки. Использовать интерфейс I2C.

Оформление курсовой работы:

Расчетно-пояснительная записка на 40-50 листах формата А4.

Дата выдачи задания «__» сентября 2025 г.

Руководитель КуР

_____ Н.Ю. Рязанова

Студент

_____ Е. А. Куликов

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Аналитический раздел	6
1.1 Постановка задачи	6
1.2 Аппаратные интерфейсы взаимодействия платы и дисплея	6
1.2.1 Интерфейс GPIO	6
1.2.2 Интерфейс I2C	7
1.2.3 Выбор аппаратного интерфейса	7
1.3 Управление внешними устройствами в Linux	9
1.3.1 Файлы устройств	9
1.3.2 Типы драйверов устройств	9
1.3.3 Идентификация устройств	10
1.3.4 Выбор типа драйвера	11
1.4 Интерфейс взаимодействия пользователя и устройства	12
1.4.1 Взаимодействие через файл символьного устройства	12
1.4.2 Взаимодействие через файловую систему /proc	12
1.4.3 Выбор интерфейса взаимодействия пользователя и устройства	14
1.5 Выполнение отложенных действий в Linux	14
1.5.1 Гибкие прерывания	15
1.5.2 Таймеры ядра	15
1.5.3 Тасклеты	16
1.5.4 Очереди работ	16
1.5.5 Выбор механизма выполнения отложенных действий	17
2 Конструкторский раздел	18
2.1 Структура ПО	18

2.2 IDEF0 диаграммы	18
2.3 Схемы алгоритмов	20
2.3.1 Алгоритм обработки переданного текста	20
2.3.2 Алгоритм обработки отложенного действия прокрутки текста	21
3 Технологический раздел	22
3.1 Выбор языка программирования и среды разработки	22
3.2 Основная структура драйвера	22
3.3 Точки входа драйвера	23
3.4 Сборка и запуск	27
4 Исследовательский раздел	30
4.1 Примеры работы программы	30
ЗАКЛЮЧЕНИЕ	32
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	33
ПРИЛОЖЕНИЕ А	34

ВВЕДЕНИЕ

Контролируемое предоставление приложениям доступа к аппаратным ресурсам компьютера является фундаментальной задачей операционных систем. Механизмом для решения этой задачи в ядре Linux выступает драйвер устройства. Данная работа посвящена созданию и исследованию такого драйвера для управления периферийным LCD-дисплеем.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу, необходимо разработать драйвер для управления LCD-дисплеем на одноплатном компьютере Raspberry Pi. Необходимо предоставить пользователю драйвера возможность отображать заданный им текст на экране подключенного к плате LCD-дисплея в формате статичной или бегущей строки. Для достижения поставленной цели требуется решить следующие задачи:

- 1) провести анализ функций и структур ядра, представляющих возможность реализации разрабатываемого драйвера;
- 2) провести анализ существующих интерфейсов подключения дисплея к плате и обмена информацией между ними;
- 3) выбрать тип реализуемого драйвера;
- 4) выбрать формат взаимодействия пользователя с драйвером;
- 5) реализовать и протестировать драйвер.

1.2 Аппаратные интерфейсы взаимодействия платы и дисплея

Для управления периферийным устройством, таким как LCD-дисплей, в операционной системе доступно несколько аппаратных интерфейсов. В контексте данной работы ключевыми альтернативами являются прямое управление выводами общего назначения GPIO [1] (General Purpose Input/Output) и использование последовательной шины I2C [2] (Inter-Integrated Circuit). Выбор между ними определяет схему соединения платы и дисплея между собой, а также использование различных структур ядра при реализации драйвера.

1.2.1 Интерфейс GPIO

Прямое управление через GPIO предполагает использование отдельных контактов микроконтроллера для каждого сигнала управления целевым устройством. Для работы с LCD-дисплеем на контроллере HD44780 [3] в 4-битном режиме потребуется минимум 7 линий GPIO: 4 линии данных (D4-D7), 2 линии

управления (RS, E) и 1 линия для чтения/записи (R/W).

Программная реализация драйвера в этом случае берет на себя полную ответственность за формирование временных диаграмм, указанных в спецификации устройства [3]. Это включает в себя установку битов данных, подачу и снятие сигнала разрешения (сигнала E) с задержками в микро и наносекундах, что реализуется программным переключением состояний выводов с использованием циклов и функций активного ожидания ядра (ndelay(), udelay()).

1.2.2 Интерфейс I2C

Использование шины I2C требует подключения всего 4 сигнальных линий: SDA (данные), SCL(тактовый сигнал), VCC(питание) и GND (земля). Физическое взаимодействие с LCD осуществляется через преобразователь (например, микросхему PCF8574, которая может входить в комплект при покупке LCD–дисплея), который выступает в роли промежуточного звена. Чип преобразует последовательные данные, полученные по I2C, в сигналы для дисплея. С точки зрения ядра Linux, драйвер работает не с сигналами, а с абстракцией I2C–устройства, используя структуру ядра i2c_driver[4] для отправки и получения байтов. При этом формированием задержек и сигналов на физическом уровне занимается аппаратный контроллер I2C процессора, а не CPU, что снижает затраты процессорного времени.

1.2.3 Выбор аппаратного интерфейса

В качестве интерфейса взаимодействия платы и дисплея выбран I2C, так как он требует меньше проводов, что снижает вероятность ошибки сборки схемы, а также использует структуру ядра i2c_driver вместо непосредственной работы с пинами GPIO, что позволяет сконцентрировать усилия на разработке самого драйвера, а не на аппаратных особенностях системы. Итоговая схема подключения дисплея к плате через интерфейс I2C представлена на рис. 1.1.

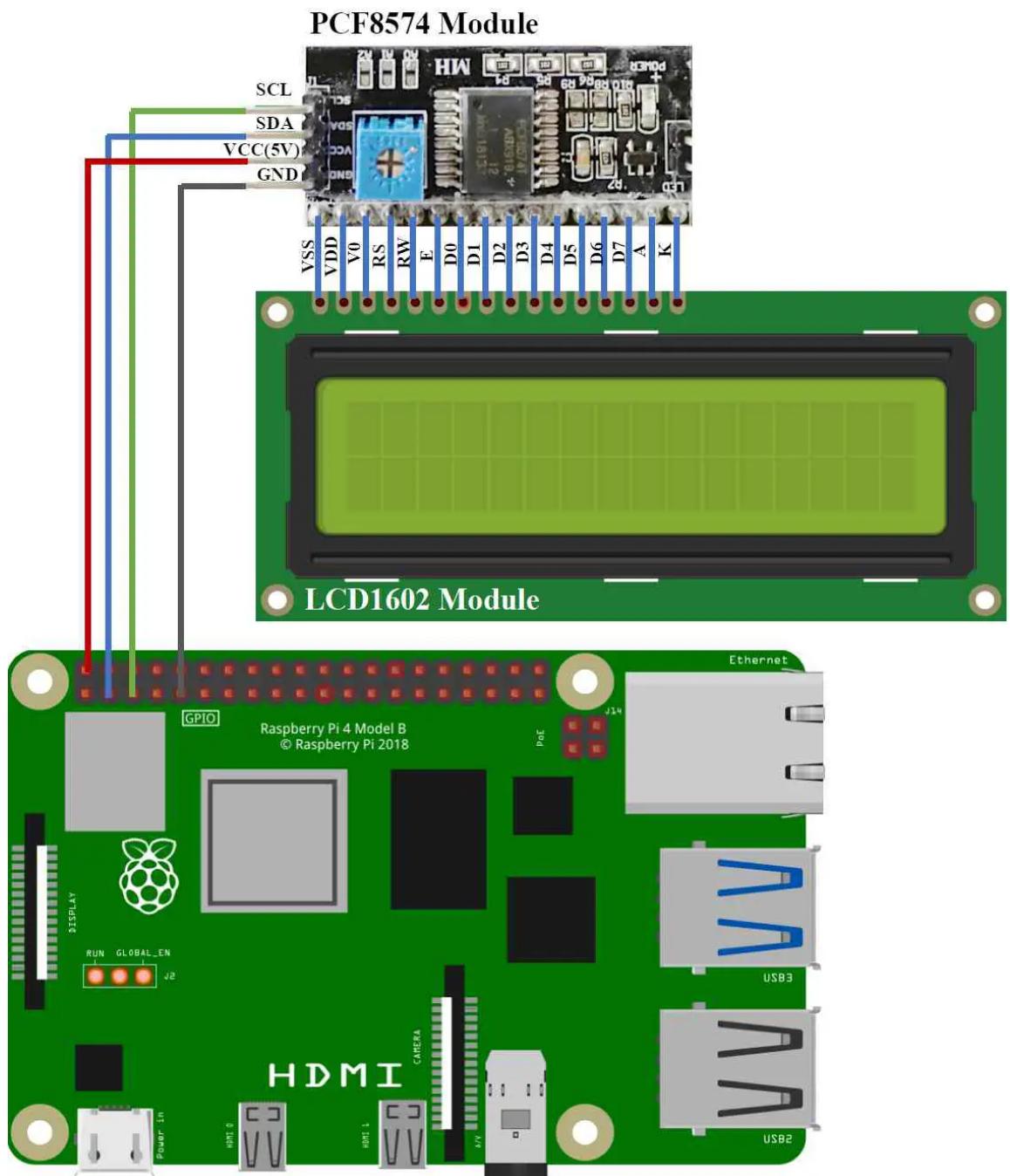


Рисунок 1.1 — Схема подключения

1.3 Управление внешними устройствами в Linux

Большая часть кода ОС связана с управлением внешними устройствами, для чего используются драйверы устройств [5].

1.3.1 Файлы устройств

ОС Unix/Linux рассматривает внешние устройства как специальные файлы. Они обеспечивают связь между файловой системой и драйверами устройств. ОС каждому устройству в соответствие ставит один специальный файл, обычно в каталоге `/dev`. Для работы с этими файлами используется структура ядра `struct file_operations`. Основные поля структуры, которые потребуются в рамках решения задачи, представлены в листинге 1.1.

Листинг 1.1 — Основные поля структуры `struct file_operations`

```
1 struct file_operations {
2     struct module *owner;
3     ...
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t
5                      *);
6     ssize_t (*write) (struct file *, const char __user *, size_t,
7                       loff_t *);
8     ...
9     int (*open) (struct inode *, struct file *);
10    ...
11 } __randomize_layout;
```

1.3.2 Типы драйверов устройств

В ОС Linux различают следующие типы устройств:

- блок-ориентированные (или блочные) устройства, например жесткий диск, передают данные блоками, все запоминающие устройства относятся к блочным;
- байт-ориентированные (или символьные) устройства (например, принтер и модем) передают данные посимвольно, как непрерывный поток байтов.

Используемый в рамках данной задачи LCD-дисплей не является устройством хранения данных и не предполагает доступ к информации по произвольным адресам. Взаимодействие с дисплеем по его спецификации [3] осуществляется путем последовательной передачи управляющих команд и символов.

Исходя из этого для реализации управления LCD-дисплеем был выбран драйвер символьного устройства. Такой тип драйвера позволяет осуществлять последовательную передачу данных, использовать стандартные операции чтения и записи, а также не требует реализации механизмов буферизации и планирования ввода-вывода, характерных для блочных устройств.

1.3.3 Идентификация устройств

Для идентификации устройств принята система major/minor номеров. Традиционно старший (major) номер идентифицирует драйвер устройства, а младший (minor) номер помогает различать устройства. В ядре Linux определен тип `typedef __kernel_dev_t dev_t` который является 32-разрядным числом. Стандарт POSIX определяет существование типа, но не формат полей, поэтому код не должен делать каких либо предположений об организации номеров внутри числа, но должен использовать макросы из `<linux/kdev_t.h>` — `MAJOR(dev_t dev)`, `MINOR(dev_t dev)`. Для новых драйверов рекомендуется использовать динамическое выделение старших номеров устройств с помощью функции `int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)`.

Существует три типа драйверов устройств в Linux [5]:

- встроенные в ядро;
- реализованные как загружаемый модуль ядра;
- разделенные между ядром и утилитой.

Устройства, соответствующие встроенным в ядро драйверам, автоматически обнаруживаются системой и становятся доступны приложениям, но для добавления такого драйвера требуется полная перекомпиляция ядра.

Драйверы, реализованные как загружаемые модули ядра, могут быть загружены или выгружены из ядра с помощью специальных команд. Процесс загрузки модуля ядра начинается в пространстве пользователя с команды `insmod`. Команда `insmod` определяет загружаемый модуль и выполняет системный вызов

`init_module`, который вызывает функцию ядра `sys_init_module`. Это основная функция загрузки модуля, которая обращается к другим функциям ядра. Аналогично, команда `rmmmod` выполняет системный вызов `delete_module`, который вызывает функцию ядра `sys_delete_module` для удаления модуля из ядра.

1.3.4 Выбор типа драйвера

В рамках данной работы для реализации управления LCD-дисплеем был выбран драйвер символьного устройства, так как взаимодействие с дисплеем по его спецификации [3] осуществляется путем последовательной передачи управляющих команд и символов.

Драйвер LCD дисплея будет реализован как загружаемый модуль ядра, так как такой подход не требует перекомпиляции ядра системы.

Так как в качестве аппаратного интерфейса взаимодействия платы и дисплея был выбран интерфейс I2C, то при реализации драйвера будет использована структура ядра `struct i2c_driver`. Основные поля структуры, которые потребуются в рамках решения задачи, представлены в листинге 1.2.

Листинг 1.2 — Основные поля структуры `struct i2c_driver`

```
1 struct i2c_driver {
2     ...
3     /* Standard driver model interfaces */
4     int (*probe)(struct i2c_client *client);
5     void (*remove)(struct i2c_client *client);
6     ...
7     struct device_driver driver;
8     const struct i2c_device_id *id_table;
9     ...
10 };
```

1.4 Интерфейс взаимодействия пользователя и устройства

Для управления разработанным драйвером LCD-дисплея необходимо предоставить пользователю механизм взаимодействия с ним из пространства пользователя. В операционной системе Linux для этих целей традиционно используются файловые интерфейсы, которые обеспечивают единый способ доступа к различным типам устройств. В рамках данной работы реализованы два интерфейса взаимодействия пользователя с драйвером устройства: через файл символьного устройства в каталоге `/dev` и через виртуальную файловую систему `/proc`.

1.4.1 Взаимодействие через файл символьного устройства

Основным и наиболее универсальным способом взаимодействия с драйвером устройства является использование файла символьного устройства, создаваемого в каталоге `/dev`.

После загрузки модуля ядра драйвер регистрирует символьное устройство, которому назначается уникальная пара `major/minor` номеров, полученная с помощью функции ядра `alloc_chrdev_region`. В пользовательском пространстве такое устройство представляется в виде файла, например `/dev/lcd0`, с которым можно работать с помощью стандартных системных вызовов `open()`, `read()` и `write()`.

Реализация данного интерфейса в драйвере осуществляется с помощью структуры ядра `struct file_operations`, в которой определяются функции-обработчики операций открытия устройства, записи данных и закрытия устройства. Основным используемым обработчиком является функция `write()`, которая получает данные из пользовательского пространства, выполняет их копирование с помощью функции `copy_from_user()`, а затем передает символы на LCD-дисплей через интерфейс I2C.

1.4.2 Взаимодействие через файловую систему `/proc`

В операционной системе Linux виртуальная файловая система `/proc` предназначена для предоставления в режиме пользователя различной информации из ядра [5]: информации о состоянии ядра, процессов и загруженных драй-

веров. В отличие от обычных файловых систем, `/proc` не хранит данные на физическом носителе, а формирует их динамически, «на лету» при обращении к соответствующим файлам [5].

Интерфейс `/proc` представляет собой механизм обмена данными между пространством пользователя и ядром, ориентированный на передачу текстовой информации и простых команд управления. Это делает его удобным инструментом для начального тестирования драйверов, отладки и демонстрации их работы, особенно в случаях, когда создание отдельного пользовательского приложения является избыточным.

Доступ к файлам в `/proc` осуществляется через специальные структуры ядра, такие как `struct proc_ops`, которые содержат указатели на функции обработки операций чтения и записи.

Файлы в файловой системе `/proc` используются для предоставления программисту или пользователю диагностической информации или отладочных данных. Взаимодействие с такими файлами осуществляется стандартными операциями чтения и записи, при этом каждая операция приводит к выполнению соответствующих функций в пространстве ядра. Однако, в отличие от интерфейса с использованием файла символьного устройства, `/proc` не в полной мере поддерживает стандартную семантику файлов, например — `ioctl`.

Системный вызов `ioctl` предназначен для передачи специфичных для устройства управляющих команд через файловый дескриптор, связанный с экземпляром устройства. Он позволяет приложениям получать доступ к функциям, уникальным для оборудования, которые обычно недоступны через абстракцию чтения/записи в файл.

Файлы, представленные в файловой системе `/proc`, являются виртуальными и не соответствуют реальным устройствам, а их обработка осуществляется через отдельный набор операций `struct proc_ops`, не предусматривающий поддержку `ioctl`.

Таким образом, файловая система `/proc` может рассматриваться как вспомогательный интерфейс взаимодействия с драйвером, ориентированный на простоту использования и наглядность, но обладающий ограничениями по функциональности по сравнению с классическими файловыми интерфейсами символьных устройств.

1.4.3 Выбор интерфейса взаимодействия пользователя и устройства

В рамках данной работы целесообразна реализация обоих интерфейсов взаимодействия пользователя и драйвера устройства.

Интерфейс символьного устройства является основным и предназначен для интеграции драйвера в различные пользовательские приложения. Он обеспечивает гибкость, расширяемость и полную функциональность.

Интерфейс `/proc`, в свою очередь, используется как вспомогательный демонстрационный механизм. Он позволяет быстро проверить работоспособность драйвера, отобразить текст на дисплее или очистить экран без написания отдельного программного обеспечения, что удобно на этапе разработки и при демонстрации работы системы.

Сочетание двух интерфейсов позволяет совместить полную функциональность драйвера с удобством его практического использования и тестирования.

1.5 Выполнение отложенных действий в Linux

При разработке драйверов устройств в ядре Linux часто возникает необходимость выполнения действий не непосредственно в момент их инициации, а с определённой задержкой либо асинхронно. Для решения данной задачи ядро предоставляет несколько механизмов, различающихся контекстом выполнения, допустимыми операциями и областью применения.

К основным механизмам выполнения отложенных действий относятся таймеры ядра, а также обработчики нижних половин — гибкие прерывания (`softirq`), тасклеты (`tasklet`) и очереди работ (`workqueues`). Каждый из указанных механизмов обладает собственными ограничениями и предназначен для определённого класса задач.

В контексте данной работы выполнение ядром отложенных действий нужно для автоматической прокрутки длинного текста, длина строки которого больше чем количество ячеек в строке дисплея.

1.5.1 Гибкие прерывания

Softirq — отложенные действия, определяемые статически во время компиляции ядра. Относятся к классу нижней половины обработчиков прерываний (bottom half). Для работы с ними используется структура ядра `softirq_action`, поля которой представлены в листинге 1.3.

Листинг 1.3 — Структура `struct softirq_action`

```
1 struct softirq_action {  
2     void (*action) (struct softirq_action *);  
3 };
```

Кроме этой структуры в ядре определен массив из 32 экземпляров данной структуры: `static struct softirq_action softirq_vec[NR_SOFT_IRQS]`.

Это означает что в ядре возможно создать 32 обработчика softirq. В настоящее время определено 10 таких обработчиков.

Так как механизм softirq выполняется в контексте прерывания, он не допускает выполнения блокирующих операций, включая захват мьютексов и операции ввода-вывода, которые могут переходить в состояние сна. Поскольку управление LCD-дисплеем осуществляется через интерфейс I2C и требует синхронизации доступа к устройству, использование softirq в рамках данного драйвера является принципиально невозможным. Кроме того, типы softirq определяются статически при компиляции ядра, что ограничивает возможность их применения в пользовательских драйверах.

1.5.2 Таймеры ядра

Таймеры ядра позволяют запланировать выполнение функции через заданный интервал времени, выраженный в джифисах. Обработчики таймеров ядра выполняются в контексте softirq [6](TIMER_SOFTIRQ), что накладывает ограничения на допустимые операции, включая запрет на использование блокирующих вызовов и механизмов синхронизации, таких как мьютессы. В связи с этим таймеры ядра не подходят для реализации отложенных операций в драйверах, использующих интерфейсы ввода-вывода, допускающие переход в состояние сна, включая I2C.

В рамках рассматриваемого драйвера таймеры не могут быть использо-

ваны напрямую, поскольку управление LCD-дисплеем осуществляется посредством шины I2C, операции которой потенциально блокируются и не должны выполняться в контексте прерывания.

1.5.3 Тасклеты

Механизм tasklet также как и механизм softirq относится к классу нижних половин обработки прерываний и используется для отложенной обработки прерываний [6]. Tasklet представляет собой один из видов softirq и упрощает их использование.

Однако обработчики tasklet выполняются в контексте softirq, что, аналогично обработчикам таймеров, запрещает использование блокирующих операций [6]. Запрещены захват мьютексов, вызовы функций ввода-вывода и любые операции, допускающие переход в состояние сна.

Поскольку в драйвере LCD-дисплея требуется выполнять операции пошине I2C и использовать мьютексы для синхронизации доступа к устройству, применение tasklet является принципиально невозможным.

1.5.4 Очереди работ

Очереди работ представляют собой механизм выполнения отложенных задач в контексте специальных потоков ядра [6]. Задачи, описываемые структурой `struct work_struct`, помещаются в очередь, описанную структурой `struct workqueue_struct` и выполняются специальными потоками-воркерами ядра.

Главным преимуществом очередей работ, в контексте данной работы и по сравнению с остальными решениями, является отсутствие ограничений на использование блокирующих операций. Код, выполняемый в очереди работ, может безопасно захватывать мьютексы, выполнять операции ввода-вывода, вызывать функции ядра и использовать задержки. Очереди работ поддерживают как немедленное, так и отложенное выполнение задач, а также позволяют управлять параметрами выполнения, включая приоритет, привязку к процессорным ядрам и максимальное число одновременно выполняемых задач [6].

1.5.5 Выбор механизма выполнения отложенных действий

В рамках данной работы будет использован механизм очередей работ для реализации отложенных действий, таких как прокрутка текста на LCD-дисплее. Управление дисплеем осуществляется через интерфейс I2C, операции которого могут блокироваться и не могут выполняться в контексте прерывания, поэтому единственным возможным решением являются очереди работ.

Выводы

В качестве интерфейса взаимодействия платы и дисплея выбран I2C, так как он требует меньше проводов, что снижает вероятность ошибки сборки схемы, а также использует структуру ядра `i2c_driver` вместо непосредственной работы с пинами GPIO, что позволяет сконцентрировать усилия на разработке самого драйвера, а не на аппаратных особенностях системы.

Для реализации управления LCD-дисплеем был выбран драйвер символьного устройства, так как взаимодействие с дисплеем по его спецификации [3] осуществляется путем последовательной передачи управляющих команд и символов. Драйвер LCD дисплея будет реализован как загружаемый модуль ядра, так как такой подход не требует перекомпиляции ядра системы.

Для обеспечения взаимодействия пользователя с устройством будут реализованы два интерфейса — через файл символьного устройства и через файл виртуальной файловой системы `/proc`. Интерфейс символьного устройства является основным и предназначен для интеграции драйвера в различные пользовательские приложения. Интерфейс `/proc`, в свою очередь, используется как вспомогательный демонстрационный механизм.

Для реализации отложенных действий таких как прокрутка текста на LCD-дисплее, будет использован механизм очередей работ так как управление дисплеем осуществляется через интерфейс I2C, операции которого могут блокироваться и не могут выполняться в контексте прерывания.

2 Конструкторский раздел

2.1 Структура ПО

Разрабатываемое ПО состоит из драйвера устройства, обращение к которому возможно через виртуальную файловую систему /proc или через файл устройства в каталоге /dev. На рис. 2.1 представлена структура ПО.

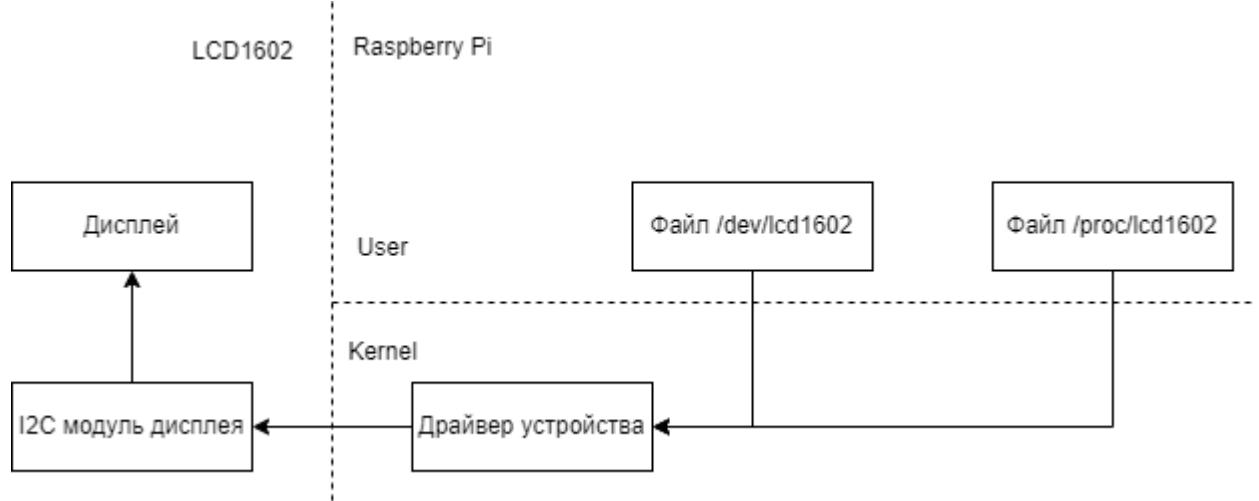


Рисунок 2.1 — Структура ПО

2.2 IDEF0 диаграммы

На рис. 2.3 — 2.2 представлены IDEF0 диаграммы нулевого и первого уровней.

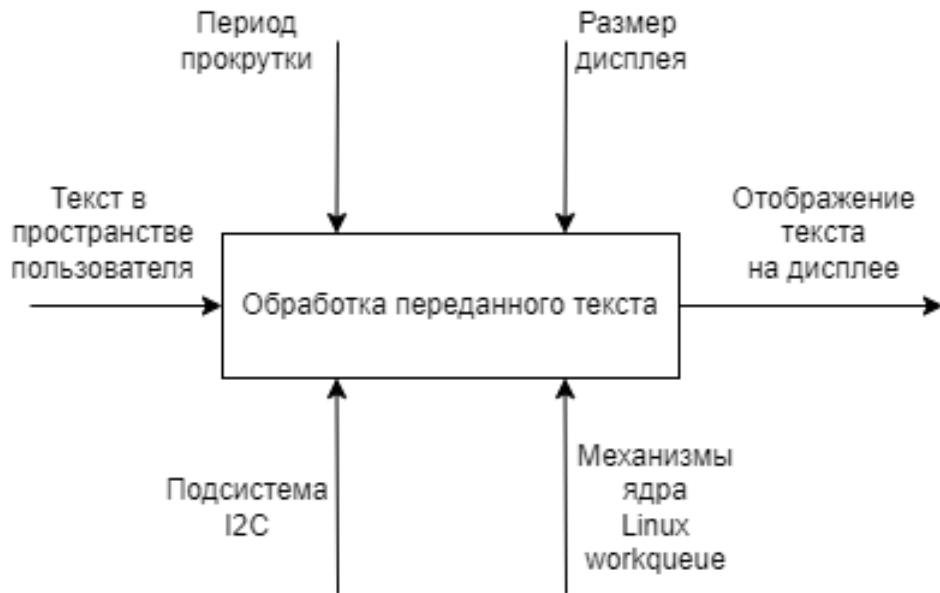


Рисунок 2.2 — Диаграмма IDEF0 нулевого уровня

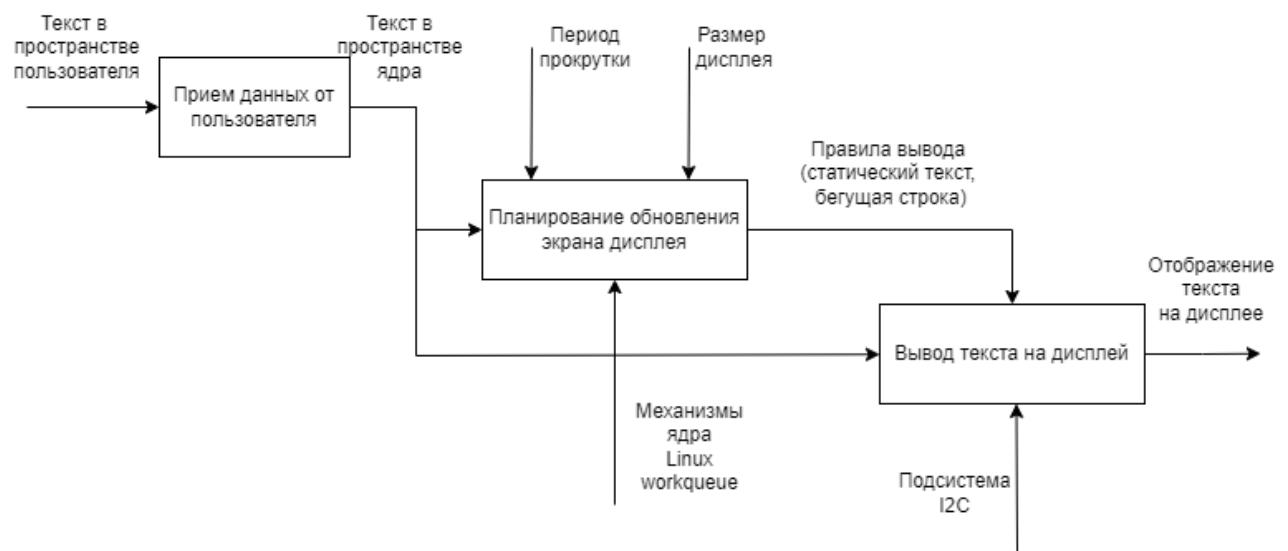


Рисунок 2.3 — Диаграмма IDEF0 первого уровня

2.3 Схемы алгоритмов

2.3.1 Алгоритм обработки переданного текста

На рис. 2.4 представлена схема работы алгоритма обработки переданного в драйвер текста.

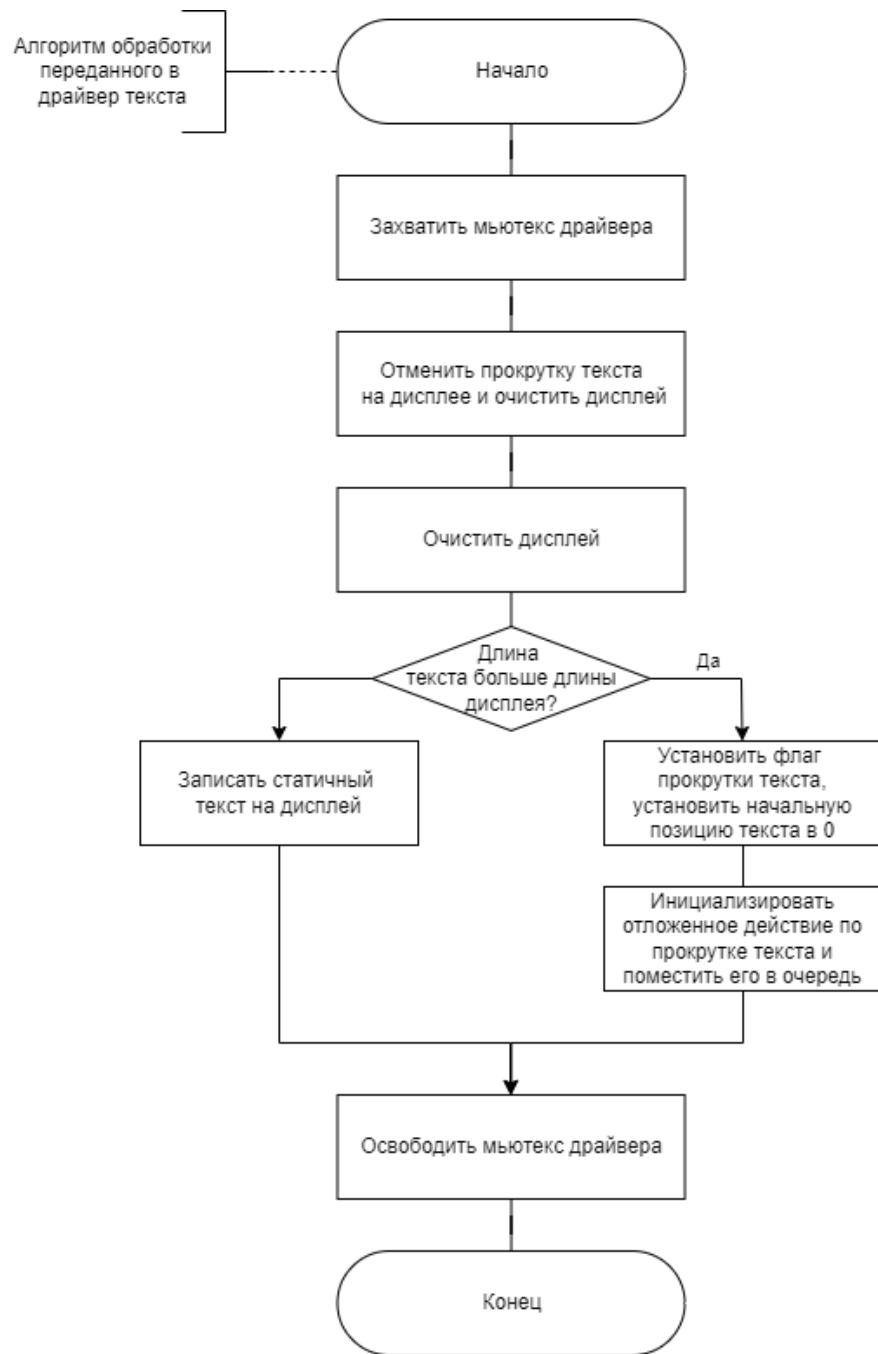


Рисунок 2.4 — Алгоритм обработки переданного текста

2.3.2 Алгоритм обработки отложенного действия прокрутки текста

На рис. 2.5 представлена схема работы алгоритма обработки отложенного действия прокрутки текста.

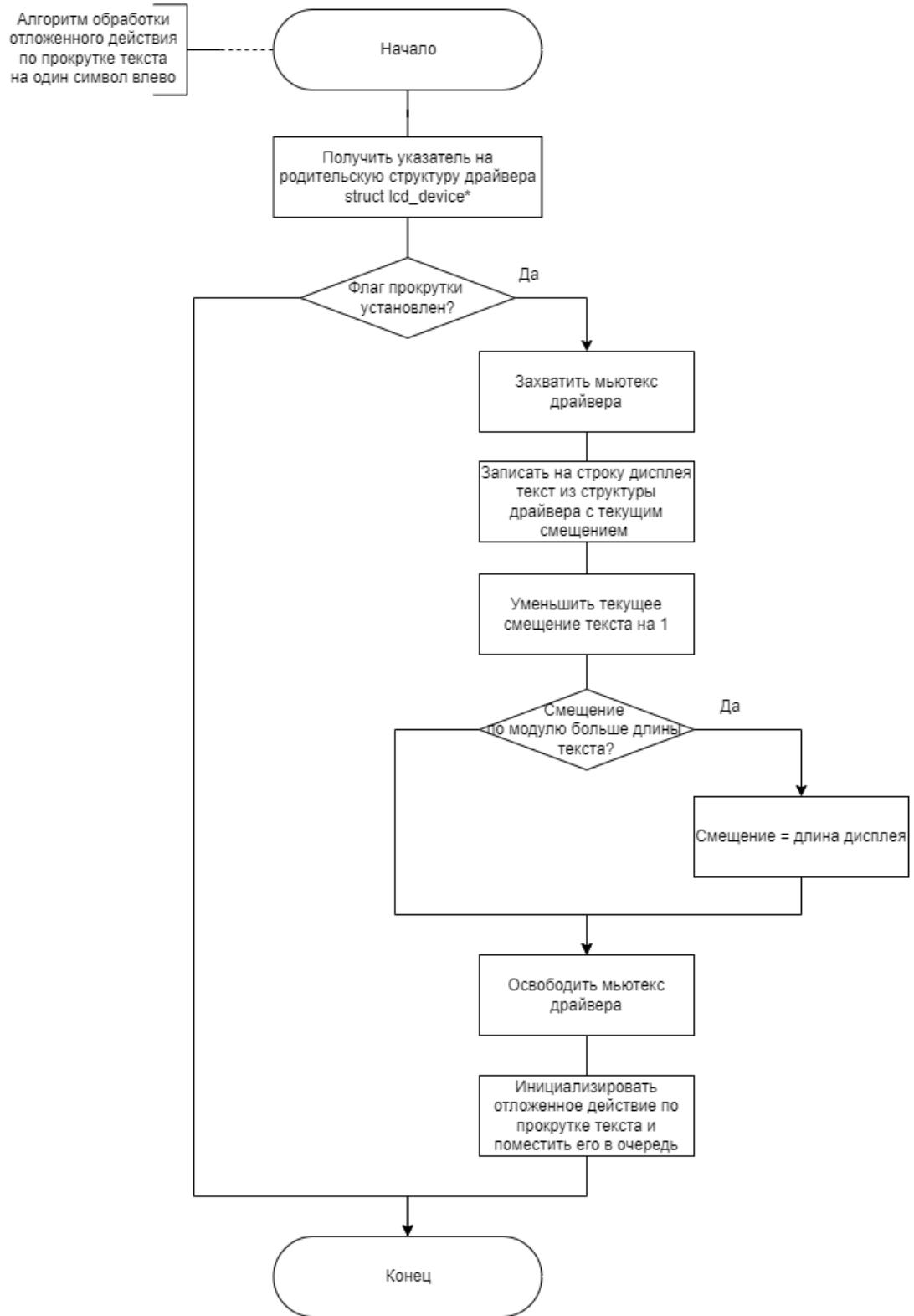


Рисунок 2.5 — Алгоритм обработки отложенного действия прокрутки текста

3 Технологический раздел

3.1 Выбор языка программирования и среды разработки

Для написания драйвера был выбран язык программирования С. Для сборки драйвера была использована утилита Make [7]. В качестве среды разработки была использована среда VSCode [8].

Разработка и тестирование драйвера выполнялись на одноплатном компьютере Raspberry Pi 3 под управлением ОС Linux с ядром версии 6.12.47+rpt-rpi-v8 (дистрибутив Raspberry Pi OS) [9].

3.2 Основная структура драйвера

Для хранения данных, связанных с подключенным LCD1602 и обеспечения работы всех его функций была объявлена структура struct lcd_device, поля которой представлены на листинге 3.1.

Листинг 3.1 — Структура struct lcd_device

```
1 struct lcd_device {
2     struct i2c_client *client;
3
4     dev_t devno;
5     struct cdev cdev;
6     struct class *class;
7     struct device *device;
8
9     struct mutex lock;
10
11    char text[LCD_HEIGHT][LCD_WIDTH + 1];
12
13    struct workqueue_struct *wq;
14    struct delayed_work scroll_work;
15
16    bool scrolling;
17    int scroll_pos;
18    char scroll_text[LCD_MAX_TEXT];
19    int scroll_delay_ms;
20};
```

3.3 Точки входа драйвера

На листингах 3.2 — 3.4 представлен код функции `probe`, вызываемой при подключении устройства.

Листинг 3.2 — Код функции `probe` часть 1

```
1 static int lcd_probe(struct i2c_client *client)
2 {
3     struct lcd_device *dev;
4     int ret;
5
6     dev = kzalloc(sizeof(*dev), GFP_KERNEL);
7     if (!dev)
8     {
9         printk(KERN_ERR "-ENOMEM error\n");
10    return -ENOMEM;
11 }
12
13 i2c_set_clientdata(client, dev);
14 dev->client = client;
15
16 mutex_init(&dev->lock);
17
18 dev->wq = alloc_workqueue("lcd1602_wq", WQ_UNBOUND |
19                           WQ_MEM_RECLAIM, 1);
20     if (!dev->wq)
21     {
22         printk(KERN_ERR "alloc_workqueue error\n");
23         ret = -ENOMEM;
24         kfree(dev);
25         return ret;
26     }
27     INIT_DELAYED_WORK(&dev->scroll_work, lcd_scroll_work);
28
29     dev->scroll_delay_ms = SCROLL_TIME_MS;
30
31     ret = alloc_chrdev_region(&dev->devno, 0, 1, DRIVER_NAME);
```

Листинг 3.3 — Код функции probe часть 2

```
1  if (ret)
2  {
3      printk(KERN_ERR "alloc_chrdev_region error\n");
4      kfree(dev);
5      destroy_workqueue(dev->wq);
6      return ret;
7  }
8
9  cdev_init(&dev->cdev, &lcd_fops);
10 cdev_add(&dev->cdev, dev->devno, 1);
11
12 dev->class = class_create(DRIVER_NAME);
13 if (IS_ERR(dev->class))
14 {
15     printk(KERN_ERR "class_create error\n");
16     ret = PTR_ERR(dev->class);
17     cdev_del(&dev->cdev);
18     unregister_chrdev_region(dev->devno, 1);
19     kfree(dev);
20     destroy_workqueue(dev->wq);
21     return ret;
22 }
23
24 dev->device = device_create(dev->class, NULL,
25                             dev->devno, NULL, "lcd0");
26 if (IS_ERR(dev->device))
27 {
28     printk(KERN_ERR "device_create error\n");
29     ret = PTR_ERR(dev->device);
30     class_destroy(dev->class);
31     cdev_del(&dev->cdev);
32     unregister_chrdev_region(dev->devno, 1);
33     kfree(dev);
34     destroy_workqueue(dev->wq);
35     return ret;
36 }
```

Листинг 3.4 — Код функции probe часть 3

```
1    lcd_proc = proc_create_data(PROC_NAME, 0666, NULL, &
2        lcd_proc_ops, dev);
3
4    if (!lcd_proc)
5    {
6        ret = -ENOMEM;
7        printk(KERN_ERR "proc create error\n");
8        device_destroy(dev->class, dev->devno);
9        class_destroy(dev->class);
10       cdev_del(&dev->cdev);
11       unregister_chrdev_region(dev->devno, 1);
12       kfree(dev);
13       destroy_workqueue(dev->wq);
14       return ret;
15   }
16
17   lcd_hw_init(dev);
18   lcd_clear(dev);
19
20   lcd_write_line(dev, 0, "LCD READY", 0);
21   lcd_write_line(dev, 1, "Driver OK", 0);
22
23   printk(KERN_INFO "module initialized successfully\n");
24
}
```

На листинге 3.5 представлен код функции `remove`, вызываемой при отключении устройства.

Листинг 3.5 — Код функции `remove`

```
1 static void lcd_remove(struct i2c_client *client)
2 {
3     struct lcd_device *dev = i2c_get_clientdata(client);
4
5     dev->scrolling = false;
6     cancel_delayed_work_sync(&dev->scroll_work);
7     destroy_workqueue(dev->wq);
8
9     lcd_clear(dev);
10
11    device_destroy(dev->class, dev->devno);
12    class_destroy(dev->class);
13    cdev_del(&dev->cdev);
14    unregister_chrdev_region(dev->devno, 1);
15    proc_remove(lcd_proc);
16    kfree(dev);
17
18    printk(KERN_INFO "module removed successfully\n");
19 }
```

Для обеспечения возможности связывания устройства с именем `lcd1602` с драйвером был инициализирован экземпляр структуры `struct i2c_device_id` — таблицы поддерживаемых устройств, а также использован макрос библиотеки `<linux/i2c.h>` — `MODULE_DEVICE_TABLE(i2c, lcd_id)`. Код инициализации структуры `struct i2c_device_id` и вызова макрона представлен на листинге 3.6.

Листинг 3.6 — Код инициализации структуры `struct i2c_device_id`

```
1 static const struct i2c_device_id lcd_id[] = {
2     { "lcd1602", 0 },
3     { }
4 };
5 MODULE_DEVICE_TABLE(i2c, lcd_id);
```

Для регистрации драйвера был проинициализирован экземпляр структуры struct i2c_driver, а также использован макрос module_i2c_driver. Код инициализации этой структуры и вызова макроса представлен на листинге 3.7.

Листинг 3.7 — Код регистрации драйвера

```
1 static struct i2c_driver lcd_driver = {  
2     .driver =  
3     {  
4         .name = DRIVER_NAME,  
5     },  
6     .probe = lcd_probe,  
7     .remove = lcd_remove,  
8     .id_table = lcd_id,  
9 };  
10  
11 module_i2c_driver(lcd_driver);
```

3.4 Сборка и запуск

Для сборки проекта была использована утилита Make и написан Makefile, код которого представлен на листингах 3.8 — 3.9.

Листинг 3.8 — Код Makefile (часть 1)

```
1 obj-m := lcd_driver.o  
2  
3 lcd_driver_v2-y := \  
4     lcd_hw.o \  
5     lcd_scroll.o \  
6     lcd_ioctl.o \  
7     lcd_proc.o \  
8     lcd_chrdev.o \  
9     lcd_main.o  
10  
11 KDIR := /lib/modules/$(shell uname -r)/build  
12 PWD := $(CURDIR)  
13  
14 I2C_BUS := i2c-1  
15 I2C_ADDR := 0x27  
16 I2C_NAME := lcd1602  
17 DRV_NAME := lcd_driver
```

Листинг 3.9 — Код Makefile (часть 2)

```
1 all:
2     $(MAKE) -C $(KDIR) M=$(PWD) modules
3
4 clean:
5     $(MAKE) -C $(KDIR) M=$(PWD) clean
6
7 load: all
8     @echo "Loading driver..."
9     sudo insmod $(DRV_NAME).ko
10
11    @echo "Creating I2C device $(I2C_NAME) at $(I2C_ADDR) on $(I2C_BUS)"
12    echo $(I2C_NAME) $(I2C_ADDR) | sudo tee /sys/bus/i2c/devices/$(I2C_BUS)/new_device
13
14 unload:
15    @echo "Removing I2C device from $(I2C_BUS)"
16    echo $(I2C_ADDR) | sudo tee /sys/bus/i2c/devices/$(I2C_BUS)/delete_device
17
18    @echo "Unloading driver..."
19    sudo rmmod $(DRV_NAME)
20
21 reload: unload load
```

Выполнение команды `make` без параметров приведет к сборке файла загружаемого модуля драйвера `lcd_driver.ko`. Для загрузки модуля используется команда `sudo insmod lcd_driver.ko`. Для выгрузки модуля используется команда `sudo rmmod lcd_driver`.

Поскольку разрабатываемый драйвер реализован как драйвер подсистемы I2C, создание экземпляра устройства и привязка его к драйверу выполняются через интерфейс `sysfs`. Для этого используется специальный файл `/sys/bus/i2c/devices/i2c-1/new_device`, в который записываются имя устройства, соответствующее идентификатору в таблице `i2c_device_id` драйвера, и I2C-адрес устройства. Для удаления экземпляра устройства и отвязывания его от драйвера используется файл `/sys/bus/i2c/devices/i2c-1/delete_device`. Используемые команды представлены на листинге 3.10.

Листинг 3.10 — Команды создания и удаления экземпляра устройства

```
1 echo lcd1602 0x27 | tee /sys/bus/i2c/devices/i2c-1/new_device  
2 echo lcd1602 | tee /sys/bus/i2c/devices/i2c-1/delete_device
```

Для удобства использования в Makefile были также реализованы цели `load`, `unload` и `reload` для автоматической сборки модуля, загрузки/выгрузки его из системы, а также создания/удаления экземпляров устройства в правильном порядке.

4 Исследовательский раздел

4.1 Примеры работы программы

Для проверки драйвера вводились различные строки, которые отображались на дисплее. На рисунках 4.1 — 4.2 представлены результаты отображения строк на экране дисплея.

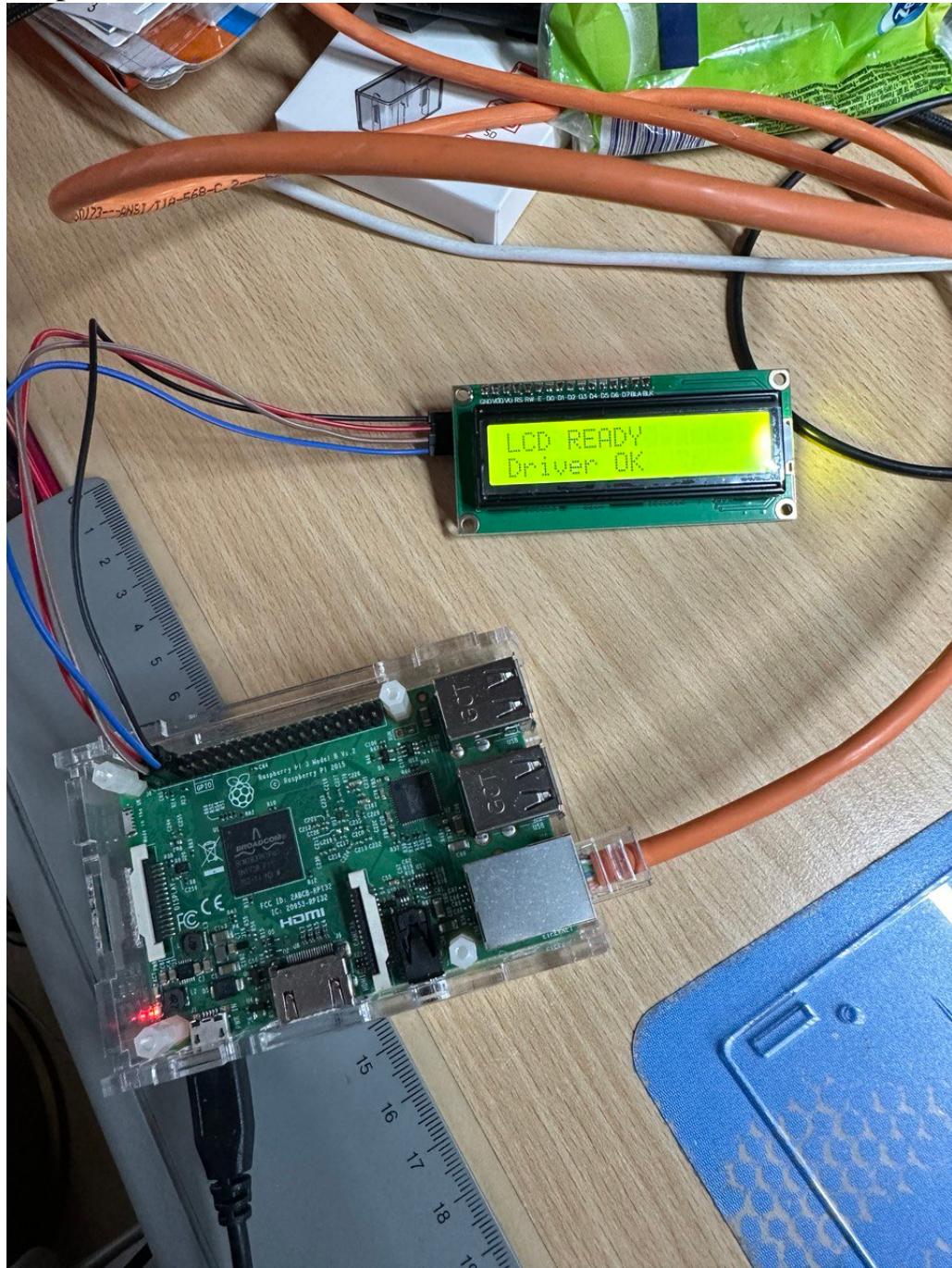


Рисунок 4.1 — Отображение на экране статичных строк при загрузке драйвера



Рисунок 4.2 — Отображение бегущей строки (один из кадров)

ЗАКЛЮЧЕНИЕ

В ходе работы был разработан драйвер для LCD-дисплея на одноплатном компьютере Raspberry Pi. Был выполнен анализ поставленной задачи, проведен сравнительный анализ методов и средств ее решения, изучен аппаратный и программный интерфейс I2C для работы с LCD-дисплеем на контроллере PCF8574, описаны использованные структуры ядра.

Драйвер был протестирован с использованием реальной платы Raspberry Pi 3B, а также дисплея LCD1602. Была также собрана электронная схема для обеспечения работы аппаратной части проекта. Тестирование работы драйвера показало, что он позволяет пользователю через интерфейсы /proc и /dev отправлять на дисплей текст, который впоследствии корректно отображается на экране в формате статичной или бегущей строки.

Поставленная цель достигнута, все задачи выполнены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация GPIO; [Электронный ресурс]. Режим доступа: <https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html> (дата обращения: 19.10.2025).
2. Документация I2C; [Электронный ресурс]. Режим доступа: <https://docs.kernel.org/i2c/index.html> (дата обращения: 19.10.2025).
3. Спецификация к LCD–дисплею 1602A; [Электронный ресурс]. Режим доступа: https://files.waveshare.com/upload/4/4d/LCD1602_I2C_Module.pdf (дата обращения: 19.10.2025).
4. Код ядра Linux; [Электронный ресурс]. Режим доступа: <https://elixir.bootlin.com/linux/v6.12.6/source> (дата обращения: 19.10.2025).
5. Вахалия Ю. UNIX изнутри // Россия, ЗАО Издательский Дом «Питер». — 2003.
6. Обзор механизмов отложенной работы; [Электронный ресурс]. Режим доступа: https://linux-kernel-labs.github.io/refs/pull/164/merge/labs/deferred_work.html (дата обращения: 10.11.2025).
7. Документация Make; [Электронный ресурс]. Режим доступа: <https://www.gnu.org/software/make/manual/make.html> (дата обращения: 20.11.2025)
8. Документация VSCode; [Электронный ресурс]. Режим доступа: <https://code.visualstudio.com/> (дата обращения: 20.11.2025)
9. RaspberryPi OS; [Электронный ресурс]. Режим доступа: <https://github.com/raspberrypi/linux> (дата обращения: 20.11.2025)

ПРИЛОЖЕНИЕ А

Листинг 4.1 — Основной заголовочный файл (часть 1)

```
1 #ifndef LCD_DRIVER_H
2 #define LCD_DRIVER_H
3
4 #include <linux/kernel.h>
5 #include <linux/i2c.h>
6 #include <linux/cdev.h>
7 #include <linux/device.h>
8 #include <linux/mutex.h>
9 #include <linux/workqueue.h>
10 #include <linux/types.h>
11
12 #define LCD_WIDTH    16
13 #define LCD_HEIGHT   2
14 #define LCD_MAX_TEXT 64
15
16 struct lcd_device {
17     struct i2c_client *client;
18
19     dev_t devno;
20     struct cdev cdev;
21     struct class *class;
22     struct device *device;
23
24     struct mutex lock;
25
26     char text[LCD_HEIGHT][LCD_WIDTH + 1];
27
28     struct workqueue_struct *wq;
29     struct delayed_work scroll_work;
30
31     bool scrolling;
32     int scroll_pos;
33     char scroll_text[LCD_MAX_TEXT];
34     int scroll_delay_ms;
35 };
```

Листинг 4.2 — Основной заголовочный файл (часть 1)

```
1 // hardware
2 void lcd_write_line(struct lcd_device *dev, int line, const
3     char *str, int offset);
4 void lcd_clear(struct lcd_device *dev);
5 void lcd_hw_init(struct lcd_device *dev);
6
7 // ioctl
8 long lcd_ioctl(struct file *file, unsigned int cmd, unsigned
9     long arg);
10
11 // scrolling
12 void lcd_scroll_work(struct work_struct *work);
13 void lcd_handle_text(struct lcd_device *dev, const char *text);
14
15 // chrdev
16 int lcd_open(struct inode *inode, struct file *file);
17 ssize_t lcd_write(struct file *f, const char __user *buf,
18     size_t len, loff_t *off);
19
20 // proc
21 ssize_t lcd_proc_write(struct file *file, const char __user *
22     buf, size_t len, loff_t *ppos);
23 int lcd_proc_show(struct seq_file *m, void *v);
24 int lcd_proc_open(struct inode *inode, struct file *file);
25
26 #endif
```

Листинг 4.3 — Код Makefile

```
1 obj-m := lcd_driver.o
2 lcd_driver_v2-y := \
3     lcd_hardware.o \
4     lcd_scroll.o \
5     lcd_ioctl.o \
6     lcd_proc.o \
7     lcd_chrdev.o \
8     lcd_main.o
9 KDIR := /lib/modules/$(shell uname -r)/build
10 PWD := $(CURDIR)
11 I2C_BUS := i2c-1
12 I2C_ADDR := 0x27
13 I2C_NAME := lcd1602
14 DRV_NAME := lcd_driver
15 all:
16     $(MAKE) -C $(KDIR) M=$(PWD) modules
17 clean:
18     $(MAKE) -C $(KDIR) M=$(PWD) clean
19 load: all
20     @echo "Loading driver..."
21     sudo insmod $(DRV_NAME).ko
22     @echo "Creating I2C device $(I2C_NAME) at $(I2C_ADDR) on $(I2C_BUS)"
23     echo $(I2C_NAME) $(I2C_ADDR) | sudo tee /sys/bus/i2c/
24         devices/$(I2C_BUS)/new_device
25 unload:
26     @echo "Removing I2C device from $(I2C_BUS)"
27     echo $(I2C_ADDR) | sudo tee /sys/bus/i2c/devices/$(I2C_BUS)
28         /delete_device
29     @echo "Unloading driver..."
30     sudo rmmod $(DRV_NAME)
31 reload: unload load
```

Листинг 4.4 — Низкоуровневые функции работы с дисплеем по I2C (часть 1)

```
1 #include <linux/i2c.h>
2 #include <linux/delay.h>
3 #include "lcd_main.h"
4 /* PCF8574 */
5 #define LCD_RS 0x01
6 #define LCD_E 0x04
7 #define LCD_BL 0x08
8 /* LCD */
9 #define LCD_CLEAR 0x01
10 #define LCD_ENTRY 0x06
11 #define LCD_ON 0x0C
12 #define LCD_FUNC 0x28
13 #define LCD_DDRAM 0x80
14 static void lcd_write_byte(struct lcd_device *dev, u8 data)
15 {
16     int ret = i2c_smbus_write_byte(dev->client, data);
17     if (ret < 0)
18         printk(KERN_INFO "I2C error\n");
19 }
20 static void lcd_write_nibble(struct lcd_device *dev, u8 nibble,
21                             u8 rs)
22 {
23     u8 data = (nibble & 0xF0) | rs | LCD_BL;
24     lcd_write_byte(dev, data | LCD_E);
25     udelay(1);
26     lcd_write_byte(dev, data & ~LCD_E);
27     udelay(50);
28 }
29 static void lcd_cmd(struct lcd_device *dev, u8 cmd)
30 {
31     lcd_write_nibble(dev, cmd, 0);
32     lcd_write_nibble(dev, cmd << 4, 0);
33 }
34 static void lcd_data(struct lcd_device *dev, u8 data)
35 {
36     lcd_write_nibble(dev, data, LCD_RS);
37     lcd_write_nibble(dev, data << 4, LCD_RS);
38 }
```

Листинг 4.5 — Низкоуровневые функции работы с дисплеем по I2C (часть 2)

```
1 void lcd_write_line(struct lcd_device *dev, int line, const
2   char *str, int offset)
3 {
4   char c;
5   int len = (str == NULL) ? 0 : strlen(str);
6   lcd_set_cursor(dev, 0, line);
7   if ((offset < 0 && -offset >= len) || offset >= LCD_WIDTH) {
8     for (int i = 0; i < LCD_WIDTH; i++) {
9       lcd_data(dev, ' ');
10      dev->text[line][i] = ' ';
11    }
12  } else if (offset >= 0)
13  {
14    for (int i = 0; i < LCD_WIDTH; i++)
15    {
16      if (i - offset >= 0 && i - offset < len)
17        c = str[i - offset];
18      else
19        c = ' ';
20      lcd_data(dev, c);
21      dev->text[line][i] = c;
22    }
23  }
24  else
25  {
26    for (int i = 0; i < LCD_WIDTH; i++)
27    {
28      if (i - offset < len)
29        c = str[i - offset];
30      else
31        c = ' ';
32      lcd_data(dev, c);
33      dev->text[line][i] = c;
34    }
35  }
36  dev->text[line][LCD_WIDTH] = '\0';
37 }
```

Листинг 4.6 — Низкоуровневые функции работы с дисплеем по I2C (часть 3)

```
1 static void lcd_set_cursor(struct lcd_device *dev, int x, int y
2     )
3 {
4     u8 addr = (y == 0) ? 0x00 : 0x40;
5     lcd_cmd(dev, LCD_DDRAM | (addr + x));
6 }
7 void lcd_clear(struct lcd_device *dev)
8 {
9     lcd_cmd(dev, LCD_CLEAR);
10    msleep(2);
11
12    lcd_write_line(dev, 0, "", 0);
13    lcd_write_line(dev, 1, "", 0);
14 }
15 void lcd_hw_init(struct lcd_device *dev)
16 {
17     msleep(50);
18     // datasheet commands sequence
19     lcd_write_nibble(dev, 0x30, 0);
20     msleep(5);
21     lcd_write_nibble(dev, 0x30, 0);
22     udelay(150);
23     lcd_write_nibble(dev, 0x30, 0);
24     udelay(150);
25
26     // set lcd to 4bit mode
27     lcd_write_nibble(dev, 0x20, 0);
28     udelay(150);
29
30     lcd_cmd(dev, LCD_FUNC);
31     lcd_cmd(dev, LCD_ON);
32     lcd_cmd(dev, LCD_ENTRY);
33     lcd_cmd(dev, LCD_CLEAR);
34     msleep(2);
35 }
```

Листинг 4.7 — Функции обработки ioctl (часть 1)

```
1 #include <linux/ioctl.h>
2 #include <linux/fs.h>
3 #include <linux/uaccess.h>
4 #include "lcd_main.h"
5
6 #define LCD_IOC_MAGIC 'L'
7
8 #define LCD_IOCTL_CLEAR _IO(LCD_IOC_MAGIC, 0)
9 #define LCD_IOCTL_SCROLL_ON _IO(LCD_IOC_MAGIC, 1)
10 #define LCD_IOCTL_SCROLL_OFF _IO(LCD_IOC_MAGIC, 2)
11 #define LCD_IOCTL_SET_SCROLL_DELAY_MS _IOW(LCD_IOC_MAGIC, 3,
12     unsigned int)
13
14 long lcd_ioctl(struct file *file, unsigned int cmd, unsigned
15     long arg)
16 {
17     struct lcd_device *dev = file->private_data;
18     unsigned int delay;
19     mutex_lock(&dev->lock);
20     switch (cmd) {
21         case LCD_IOCTL_CLEAR:
22             dev->scrolling = false;
23             cancel_delayed_work_sync(&dev->scroll_work);
24             lcd_clear(dev);
25             break;
26         case LCD_IOCTL_SCROLL_ON:
27             if (strlen(dev->scroll_text) > 0)
28             {
29                 dev->scrolling = true;
30                 queue_delayed_work(dev->wq, &dev->scroll_work,
31                     msecs_to_jiffies(dev->scroll_delay_ms));
32             }
33             break;
34         case LCD_IOCTL_SCROLL_OFF:
35             dev->scrolling = false;
36             cancel_delayed_work_sync(&dev->scroll_work);
37             break;
38     }
```

Листинг 4.8 — Функции обработки ioctl (часть 1)

```
1  case LCD_IOCTL_SET_SCROLL_DELAY_MS:
2      if (copy_from_user(&delay, (unsigned int __user *)arg,
3                         sizeof(delay)))
4      {
5          mutex_unlock(&dev->lock);
6          return -EFAULT;
7      }
8
9      if (delay < 100 || delay > 5000)
10     {
11         mutex_unlock(&dev->lock);
12         return -EINVAL;
13     }
14
15     dev->scroll_delay_ms = delay;
16     break;
17 default:
18     mutex_unlock(&dev->lock);
19     return -ENOTTY;
20 }
21 mutex_unlock(&dev->lock);
22 return 0;
}
```

Листинг 4.9 — Функция и работа для прокрутки текста

```
1 #include <linux/string.h>
2 #include <linux/workqueue.h>
3 #include <linux/mutex.h>
4 #include "lcd_main.h"
5 void lcd_scroll_work(struct work_struct *work)
6 {
7     struct lcd_device *dev = container_of(work, struct lcd_device
8         , scroll_work.work);
9     int len = strlen(dev->scroll_text);
10    if (!dev->scrolling)
11        return;
12    mutex_lock(&dev->lock);
13    lcd_write_line(dev, 0, dev->scroll_text, dev->scroll_pos);
14    dev->scroll_pos--;
15    if (dev->scroll_pos <= -len)
16        dev->scroll_pos = LCD_WIDTH;
17    mutex_unlock(&dev->lock);
18    queue_delayed_work(dev->wq, &dev->scroll_work,
19        msecs_to_jiffies(dev->scroll_delay_ms));
20 }
21 void lcd_handle_text(struct lcd_device *dev, const char *text)
22 {
23     mutex_lock(&dev->lock);
24     dev->scrolling = false;
25     cancel_delayed_work_sync(&dev->scroll_work);
26     lcd_clear(dev);
27     if (strlen(text) > LCD_WIDTH)
28     {
29         strscpy(dev->scroll_text, text, LCD_MAX_TEXT);
30         dev->scroll_pos = 0;
31         dev->scrolling = true;
32         queue_delayed_work(dev->wq, &dev->scroll_work,
33             msecs_to_jiffies(dev->scroll_delay_ms));
34     }
35 }
```

Листинг 4.10 — Функции работы с char device

```
1 #include <linux/fs.h>
2 #include <linux/uaccess.h>
3 #include "lcd_main.h"
4 ssize_t lcd_write(struct file *f, const char __user *buf,
5                   size_t len, loff_t *off)
6 {
7     struct lcd_device *dev = f->private_data;
8     char kbuf[LCD_MAX_TEXT];
9
10    if (len == 0)
11        return 0;
12
13    if (len >= LCD_MAX_TEXT)
14        len = LCD_MAX_TEXT - 1;
15
16    if (copy_from_user(kbuf, buf, len))
17        return -EFAULT;
18
19    kbuf[len] = '\0';
20    if (kbuf[len - 1] == '\n')
21        kbuf[len - 1] = '\0';
22
23    lcd_handle_text(dev, kbuf);
24    return len;
25}
26 int lcd_open(struct inode *inode, struct file *file)
27 {
28     file->private_data = container_of(inode->i_cdev, struct
29         lcd_device, cdev);
30     return 0;
31 }
```

Листинг 4.11 — Функции работы с proc

```
1 #include <linux/proc_fs.h>
2 #include <linux/seq_file.h>
3 #include <linux/uaccess.h>
4 #include "lcd_main.h"
5 ssize_t lcd_proc_write(struct file *file, const char __user *
6     buf, size_t len, loff_t *ppos)
7 {
8     struct lcd_device *dev = pde_data(file_inode(file));
9     char kbuf[LCD_MAX_TEXT];
10    if (len == 0)
11        return 0;
12    if (len >= LCD_MAX_TEXT)
13        len = LCD_MAX_TEXT - 1;
14    if (copy_from_user(kbuf, buf, len))
15        return -EFAULT;
16    kbuf[len] = '\0';
17    if (kbuf[len - 1] == '\n')
18        kbuf[len - 1] = '\0';
19    lcd_handle_text(dev, kbuf);
20    return len;
21 }
22 int lcd_proc_show(struct seq_file *m, void *v)
23 {
24     struct lcd_device *dev = m->private;
25     mutex_lock(&dev->lock);
26     seq_printf(m, "LCD1602\n");
27     seq_printf(m, "Scrolling: %s\n", dev->scrolling ? "yes" : "no");
28     seq_printf(m, "Line 0: %s\n", dev->text[0]);
29     seq_printf(m, "Line 1: %s\n", dev->text[1]);
30     mutex_unlock(&dev->lock);
31     return 0;
32 }
33 int lcd_proc_open(struct inode *inode, struct file *file)
34 {
35     return single_open(file, lcd_proc_show, pde_data(inode));
}
```

Листинг 4.12 — Основной файл драйвера (часть 1)

```
1  /*
2   * LCD1602 I2C Driver for Raspberry Pi
3   * HD44780 + PCF8574
4   */
5  #include <linux/module.h>
6  #include <linux/init.h>
7  #include <linux/kernel.h>
8  #include <linux/i2c.h>
9  #include <linux/slab.h>
10 #include <linux/delay.h>
11 #include <linux/fs.h>
12 #include <linux/cdev.h>
13 #include <linux/device.h>
14 #include <linux/proc_fs.h>
15 #include "lcd_main.h"
16 #define DRIVER_NAME "lcd1602"
17 #define SCROLL_TIME_MS 1000
18 #define PROC_NAME "lcd1602"
19 static struct proc_dir_entry *lcd_proc;
20 static const struct proc_ops lcd_proc_ops = {
21     .proc_open    = lcd_proc_open,
22     .proc_read    = seq_read,
23     .proc_lseek   = seq_lseek,
24     .proc_release = single_release,
25     .proc_write   = lcd_proc_write,
26 };
27 static const struct file_operations lcd_fops = {
28     .owner = THIS_MODULE,
29     .open  = lcd_open,
30     .write = lcd_write,
31     .unlocked_ioctl = lcd_ioctl,
32 };
```

Листинг 4.13 — Основной файл драйвера (часть 2)

```
1 static int lcd_probe(struct i2c_client *client)
2 {
3     struct lcd_device *dev;
4     int ret;
5     dev = kzalloc(sizeof(*dev), GFP_KERNEL);
6     if (!dev)
7     {
8         printk(KERN_ERR "-ENOMEM error\n");
9         return -ENOMEM;
10    }
11    i2c_set_clientdata(client, dev);
12    dev->client = client;
13    mutex_init(&dev->lock);
14    dev->wq = alloc_workqueue("lcd1602_wq", WQ_UNBOUND |
15        WQ_MEM_RECLAIM, 1);
16    if (!dev->wq)
17    {
18        printk(KERN_ERR "alloc_workqueue error\n");
19        ret = -ENOMEM;
20        kfree(dev);
21        return ret;
22    }
23    INIT_DELAYED_WORK(&dev->scroll_work, lcd_scroll_work);
24    dev->scroll_delay_ms = SCROLL_TIME_MS;
25    ret = alloc_chrdev_region(&dev->devno, 0, 1, DRIVER_NAME);
26    if (ret)
27    {
28        printk(KERN_ERR "alloc_chrdev_region error\n");
29        kfree(dev);
30        destroy_workqueue(dev->wq);
31        return ret;
32    }
33    cdev_init(&dev->cdev, &lcd_fops);
34    cdev_add(&dev->cdev, dev->devno, 1);
35    dev->class = class_create(DRIVER_NAME);
36    if (IS_ERR(dev->class))
37    {
38        printk(KERN_ERR "class_create error\n");
39        ret = PTR_ERR(dev->class);
40    }
41 }
```

```

39      cdev_del(&dev->cdev);
40      unregister_chrdev_region(dev->devno, 1);
41      kfree(dev);
42      destroy_workqueue(dev->wq);
43      return ret;
44 }
45 dev->device = device_create(dev->class, NULL, dev->devno,
46                             NULL, "lcd0");
47 if (IS_ERR(dev->device))
48 {
49     printk(KERN_ERR "device_create error\n");
50     ret = PTR_ERR(dev->device);
51     class_destroy(dev->class);
52     cdev_del(&dev->cdev);
53     unregister_chrdev_region(dev->devno, 1);
54     kfree(dev);
55     destroy_workqueue(dev->wq);
56     return ret;
57 }
58 lcd_proc = proc_create_data(PROC_NAME, 0666, NULL, &
59                             lcd_proc_ops, dev);
60 if (!lcd_proc)
61 {
62     ret = -ENOMEM;
63     printk(KERN_ERR "proc create error\n");
64     device_destroy(dev->class, dev->devno);
65     class_destroy(dev->class);
66     cdev_del(&dev->cdev);
67     unregister_chrdev_region(dev->devno, 1);
68     kfree(dev);
69     destroy_workqueue(dev->wq);
70     return ret;
71 }
72 lcd_hw_init(dev);
73 lcd_clear(dev);
74 lcd_write_line(dev, 0, "LCD READY", 0);
75 lcd_write_line(dev, 1, "Driver OK", 0);
76 printk(KERN_INFO "module initialized successfully\n");
77 return 0;
78 }
```

Листинг 4.14 — Основной файл драйвера (часть 3)

```
1 static void lcd_remove(struct i2c_client *client)
2 {
3     struct lcd_device *dev = i2c_get_clientdata(client);
4     dev->scrolling = false;
5     cancel_delayed_work_sync(&dev->scroll_work);
6     destroy_workqueue(dev->wq);
7     lcd_clear(dev);
8     device_destroy(dev->class, dev->devno);
9     class_destroy(dev->class);
10    cdev_del(&dev->cdev);
11    unregister_chrdev_region(dev->devno, 1);
12    proc_remove(lcd_proc);
13    kfree(dev);
14    printk(KERN_INFO "module removed successfully\n");
15 }
16 static const struct i2c_device_id lcd_id[] = {
17     { "lcd1602", 0 },
18     { }
19 };
20 MODULE_DEVICE_TABLE(i2c, lcd_id);
21 static struct i2c_driver lcd_driver = {
22     .driver =
23     {
24         .name = DRIVER_NAME,
25     },
26     .probe = lcd_probe,
27     .remove = lcd_remove,
28     .id_table = lcd_id,
29 };
30 module_i2c_driver(lcd_driver);
31 MODULE_LICENSE("GPL");
32 MODULE_AUTHOR("Egor Kulikov");
33 MODULE_DESCRIPTION("LCD1602 I2C Driver");
```