

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б15-мм

Портирование операционной системы Martos под STM32F4

Родионов Максим Александрович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
ст. преподаватель кафедры ИАС, К. К. Смирнов

Консультант:
ведущий разработчик ПО, ООО «СофтКом», И. С. Архипов

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
2.1. Основные компоненты Martos	5
2.2. Системно-зависимые функции	6
2.3. Инструменты для разработки	6
2.4. Вывод	8
3. Реализация	9
3.1. Управление периферией	9
3.2. UART	10
3.3. Управление памятью	11
3.4. Управление таймерами	11
3.5. Реализация вытесняющей многозадачности	11
3.6. Написание примеров на Rust	13
3.7. Написание примера на C	14
3.8. Запуск примеров на плате	16
4. Тестирование и апробация	18
4.1. Тестирование	18
4.2. Апробация	18
5. Заключение	19
Список литературы	20

Введение

Операционные системы являются важной частью современной вычислительной инфраструктуры, обеспечивая взаимодействие между аппаратным обеспечением и прикладными программами. Одним из ключевых аспектов разработки операционных систем является обеспечение их совместимости с различными аппаратными платформами. Это позволяет расширить сферу применения операционной системы и сделать её более доступной для разработчиков и исследователей.

Компания ООО «СофтКом» разрабатывает операционную систему Martos, которая поддерживает запуск программ на различных процессорных архитектурах, включая MIPS64, RISC-V и Xtensa. Однако в условиях растущей популярности устройств на базе архитектуры ARM [1], особенно таких доступных и широко используемых платформ, как STM32¹, возникает необходимость адаптации Martos для работы на этих устройствах. Портирование операционной системы на новые платформы может не только расширить её функциональные возможности, но и упростить её изучение.

Основной целью данной работы является портирование операционной системы Martos на серию микроконтроллеров STM32F4, оснащённую 32-битным процессором ARM Cortex-M4. Для достижения этой цели было решено использовать библиотеку аппаратных абстракций (Hardware Abstraction Layer, HAL) stm32f4xx-hal [4], разработанную сообществом для микроконтроллеров STM32F4. Данная библиотека предоставляет высокоуровневый интерфейс для взаимодействия с периферией микроконтроллера, что позволяет упростить процесс портирования и минимизировать зависимость кода от конкретной аппаратной платформы.

¹https://www.st.com/content/st_com/en.html

1. Постановка задачи

Целью данной работы является адаптация операционной системы Martos для работы на серии микроконтроллеров STM32F4. Это подразумевает портирование ключевых функций операционной системы и её последующее тестирование на целевой платформе. Для достижения поставленной цели были определены следующие задачи:

1. провести обзор основных компонентов операционной системы и доступных инструментов для разработки;
2. реализовать поддержку базовых функций операционной системы, таких как управление динамической памятью, работа с таймерами, адаптация вытесняющего планировщика задач и поддержка UART²;
3. разработать набор демонстрационных примеров на языках программирования Rust и C для проверки работоспособности основных компонентов Martos.

²Universal Asynchronous Receiver-Transmitter

2. Обзор

Для портирования операционной системы необходимо определить ключевые этапы переноса и возможные сложности, связанные с особенностями целевой платформы. В данном разделе представлен обзор основных компонентов операционной системы Martos и их взаимодействия, а также инструментов, доступных для разработки под платформу STM32F4.

2.1. Основные компоненты Martos

Операционная система Martos, написанная на языке Rust, представляет собой ОС, предназначенную для работы на встраиваемых системах. Её архитектура включает несколько ключевых компонентов, которые обеспечивают базовую функциональность:

2.1.1. Менеджер памяти

Управление памятью является одной из ключевых задач любой операционной системы, особенно в условиях ограниченных ресурсов микроконтроллеров. Для портирования Martos на STM32F4 необходимо найти и использовать библиотеку, реализующую выделение и освобождение динамической памяти, а также за выделение памяти на куче, которое адаптировано под архитектуру микроконтроллера.

2.1.2. Таймеры

Таймеры играют важную роль в современных вычислительных системах, обеспечивая точное выполнение задач по времени. Для портирования этого компонента необходимо реализовать функции взаимодействия с таймерами микроконтроллера, соответствующие API, разработанному в рамках предыдущего семестра [8].

2.1.3. Preemptive диспетчер задач

Preemptive (вытесняющий) диспетчер задач является ключевым компонентом, обеспечивающим многозадачность в Martos. Он позволяет операционной системе переключаться между задачами на основе временных интервалов. В Martos функциональность диспетчера уже реализована, для его работы на целевой платформе потребуется реализация низкоуровневых функций, таких как настройка прерываний, инициализация стека задач, а также сохранение и восстановление контекста.

2.2. Системно-зависимые функции

Таким образом, для портирования системы на целевую платформу необходимо реализовать системно-зависимые функции, интерфейсы которых уже определены в системе. Эти функции, представленные на рис. 1, будут обеспечивать взаимодействие между высокоуровневыми компонентами Martos и аппаратным обеспечением STM32F4.

2.3. Инструменты для разработки

Для разработки встраиваемых систем на языке Rust существует несколько инструментов с открытым исходным кодом, которые могут упростить процесс работы с микроконтроллерами, включая STM32F4:

2.3.1. cortex-m и cortex-m-rt

Библиотека cortex-m [3] предоставляет низкоуровневые API для работы с ядром Cortex-M. Она включает функции для управления прерываниями, работы с регистрами процессора и выполнения специфичных для Cortex-M операций.

Библиотека cortex-m-rt [6] предоставляет минимальную среду выполнения (runtime) для программ на Rust, предназначенных для микроконтроллеров Cortex-M. Она отвечает за инициализацию аппаратно-

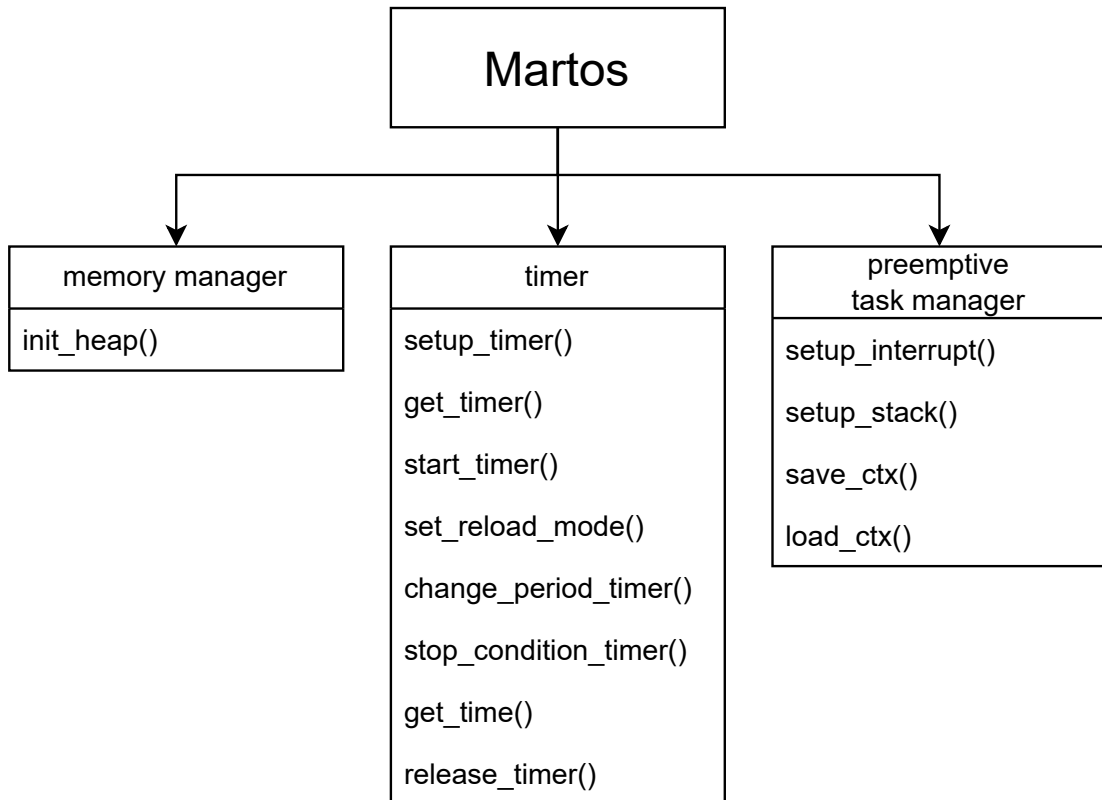


Рис. 1: Интерфейсы Martos

го обеспечения, настройку стека, обработку исключений и прерываний и запуск программы.

2.3.2. Hardware Abstraction Layer

Для работы с аппаратным обеспечением микроконтроллеров STM32F4 в экосистеме Rust существует библиотека `stm32f4xx-hal`. Это слой аппаратных абстракций (HAL), который предоставляет унифицированные интерфейсы для взаимодействия с периферийными устройствами микроконтроллера, что упрощает к ним доступ на уровне программного обеспечения.

2.3.3. embedded-alloc

Для работы с динамической памятью на процессорах Cortex-M доступна библиотека `embedded-alloc` [7]. Эта библиотека предоставляет

реализацию глобального распределителя памяти, ориентированного на использование во встраиваемых системах.

2.3.4. Прошивка и отладка: OpenOCD и GDB

Для отладки и прошивки микроконтроллеров STM32F4 используются OpenOCD (Open On-Chip Debugger) и gdb-multiarch.

OpenOCD [5] — это открытый отладчик для работы с JTAG/SWD-интерфейсами, поддерживающий множество микроконтроллеров, включая STM32. Он позволяет прошивать *firmware*, управлять выполнением программы, читать и модифицировать регистры, а также работать с памятью.

gdb-multiarch — это GNU Debugger (GDB), поддерживающий различные архитектуры, в том числе ARM Cortex-M. В связке с OpenOCD он позволяет выполнять пошаговую отладку, устанавливать точки останова, анализировать состояние регистров и памяти.

Эта связка инструментов полезна при портировании операционной системы, так как позволяет отслеживать работу ядра, выявлять ошибки в обработчиках прерываний и проверять корректность работы с аппаратными ресурсами.

2.4. Вывод

В данном обзоре рассмотрены основные компоненты Martos, обеспечивающие базовую функциональность системы, а также выделены системно-зависимые функции, необходимые для их работы на платформе STM32F4. К ним относятся настройка прерываний, управление памятью и работа с таймерами. Кроме того, представлены инструменты, которые будут использованы при портировании системы на новое аппаратное обеспечение.

3. Реализация

В данном разделе описывается реализация базовых функций операционной системы для платформы STM32F4 и разработанный набор тестовых примеров.

На рис. 2 представлена диаграмма взаимодействия компонентов.

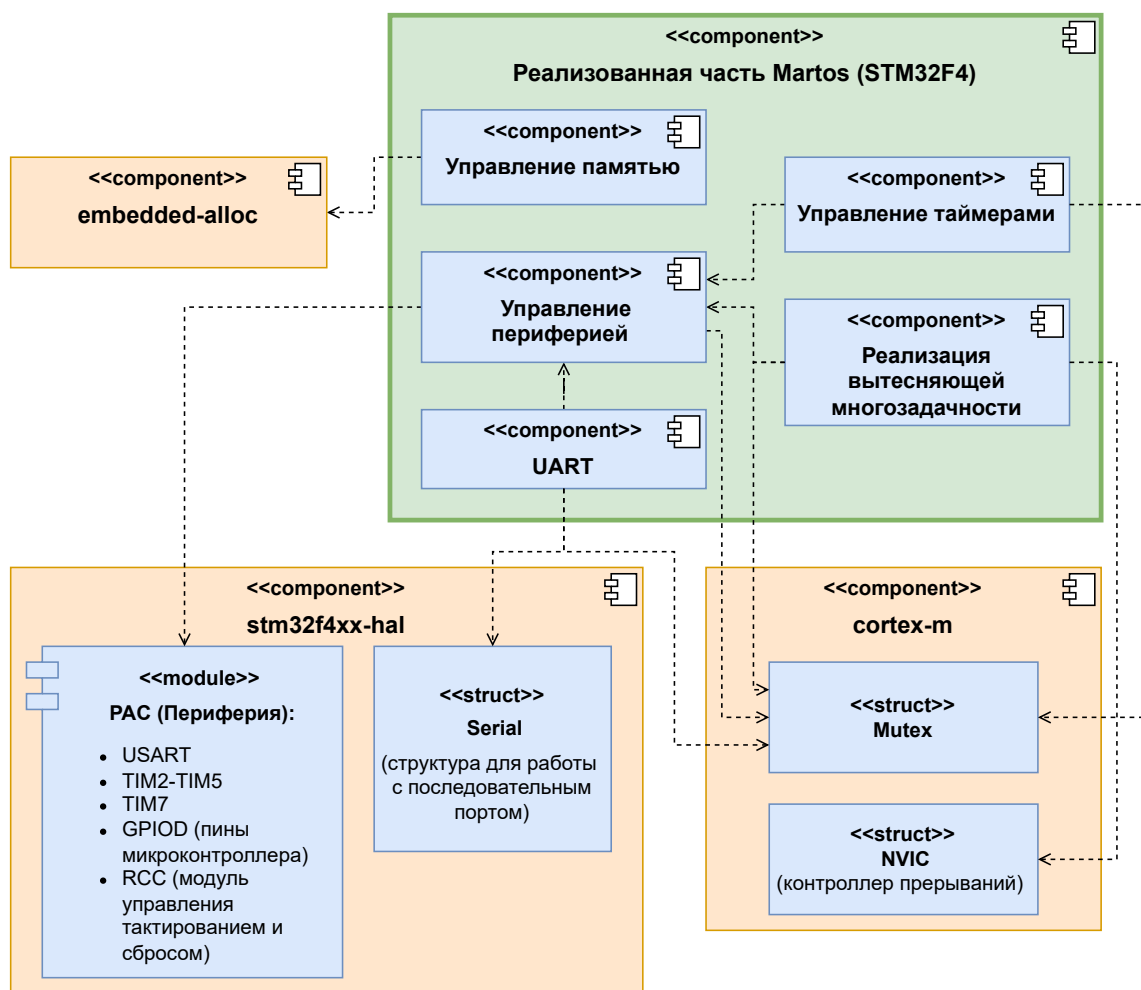


Рис. 2: Диаграмма компонентов

3.1. Управление периферией

Для работы с периферийными устройствами микроконтроллера была реализована структура `Peripherals`, которая инкапсулирует ключе-

вые аппаратные компоненты, такие как таймеры, USART³, GPIO⁴ и объект, хранящий настройки тактирования системных часов. Эта структура позволяет централизовать управление периферийными устройствами и организовать взаимодействие между различными компонентами системы.

Для обеспечения глобального доступа к экземпляру структуры `Peripherals` используется статическая переменная `PERIPHERALS`. Она представляет собой обёртку типа `Mutex<RefCell<Option<Peripherals>>>`, что позволяет безопасно работать с периферией в контексте прерываний и многозадачности. Инициализация периферии выполняется в рамках критической секции, что помогает избежать состояния гонки при работе с общими ресурсами.

3.2. UART

Для организации последовательной передачи данных посредством UART была реализована функция `get_uart()`, возвращающая настроенный экземпляр последовательного порта. В рамках этой функции соответствующие порты, переводятся в альтернативный режим, что делает их пригодными для приёма и передачи данных через USART. Для создания последовательного порта используется метод `Serial::new` из библиотеки `stm32f4xx-hal`. В него передаются: периферийное устройство USART, кортеж настроенных пинов, скорость передачи и ссылка на настройки тактирования системы. Вызываемая функция автоматически настраивает USART в асинхронный режим, соответствующий стандарту UART, и конфигурирует внутренние регистры для корректной работы.

³Universal Synchronous/Asynchronous Receiver-Transmitter

⁴general-purpose I/O

3.3. Управление памятью

Для работы с динамическим выделением памяти была использована библиотека `embedded-alloc`, предоставляющая реализацию аллокатора `LlffHeap` (Linked List First Fit heap). Инициализация кучи выполняется в функции `init_heap()`, где выделяется статический буфер фиксированного размера, который затем передаётся аллокатору для управления. С помощью атрибута `#[global_allocator]` указывается глобальный аллокатор, который будет использоваться по умолчанию при выполнении операций выделения и освобождения памяти.

3.4. Управление таймерами

Модуль `hardware_timer.rs` реализует функциональность управления таймерами, соответствующую API. Основным компонентом модуля является структура `TimerBlock`, которая управляет конфигурацией и состоянием нескольких аппаратных таймеров. Экземпляр `TimerBlock` хранится в глобальной переменной `TIMER_BLOCK`, которая аналогично `PERIPHERALS` инкапсулирована в обертку типа `Mutex<RefCell<Option<TimerBlock>>>`. Структура `TimerBlock` содержит четыре аппаратных таймера общего назначения (`TIM2`, `TIM3`, `TIM4`, `TIM5`) и связанные с ними параметры:

- массив атомарных флагов, указывающих, занят ли конкретный таймер;
- массив значений типа `Duration`, хранящих установленные периоды для каждого таймера.

3.5. Реализация вытесняющей многозадачности

Для поддержки вытесняющей многозадачности был реализован механизм управления контекстом задач. Этот механизм включает сохранение и восстановление состояния процессора, а также настройку тайме-

ра для планирования задач. Особое внимание уделяется работе с регистрами процессора Cortex-M4, особенно при использовании блока FPU⁵.

При возникновении прерывания на Cortex-M4 с включенным блоком FPU автоматически формируется стековый фрейм (stack frame), который включает следующие регистры:

- R0–R3, R12: регистры общего назначения;
- LR (R14): регистр ссылок, хранящий адреса возврата;
- PC (R15): счётчик команд;
- xPSR: регистр состояния программы;
- S0–S15: регистры для работы с числами с плавающей точкой;
- FPSCR: регистр состояния и управления числами с плавающей точкой.

Этот процесс описан в техническом руководстве для STM32F4xxx Cortex-M4 [2].

Остальные регистры, не сохраняются автоматически и должны сохраняться программно. Для этого была создана структура **TrapFrame**, которая представляет собой описание состояния процессора, сохраняемого при переключении контекста задачи. Структура включает:

- R4–R11: регистры общего назначения, которые необходимо сохранять вручную;
- S16–S31: дополнительные регистры для работы с числами с плавающей запятой;
- PSP⁶: указатель стека процесса, используемый для задач в пользовательском режиме.

Сохранение и восстановление данных регистров выполняется с помощью ассемблерных вставок.

⁵Floating-point unit

⁶Process Stack Pointer

3.6. Написание примеров на Rust

В рамках проекта были написаны примеры на языке Rust, демонстрирующие реализованную функциональность.

3.6.1. Пример hello-world

Пример проверяет базовые принципы работы многозадачной системы, включая инициализацию системы, добавление задачи в невытесняющий планировщик и управление её выполнением. Задача выполняет три основные функции:

- `setup_fn`: выполняется один раз при инициализации задачи. В данном случае она отправляет сообщение «Setup 'hello world' from UART!» через последовательный интерфейс (UART);
- `loop_fn`: циклически увеличивает значение глобального счетчика и выводит его через полу-хостинг (`hprintln!` - механизм, который позволяет плате взаимодействовать с хост-компьютером через отладочный интерфейс);
- `stop_condition_fn`: проверяет условие завершения задачи. Если значение счетчика достигает 50, задача завершается.

3.6.2. Пример dynamic-memory

Данный пример проверяет работу системы с динамической памятью. Он демонстрирует, как задачи могут использовать динамические структуры данных, такие как вектор (`Vec`), для хранения информации. Задача выполняет следующие действия: увеличивает значение глобального счетчика, добавляет текущее значение счетчика в вектор `VEC`, выводит последнее значение вектора через полу-хостинг.

3.6.3. Пример timer

Пример демонстрирует, как задачи могут использовать аппаратные таймеры для измерения времени. В функции `setup_fn` инициализиру-

ется таймер: устанавливается период таймера, таймер запускается, и ссылка на него сохраняется в глобальной переменной. В рамках циклической функции увеличивается значение глобального счётчика, извлекается текущее время из таймера, преобразуется в микросекунды и выводится полученное значение через полу-хостинг. Если значение счётчика достигает 50, задача завершается. При этом таймер останавливается, и его ресурсы освобождаются.

3.6.4. Пример scheduler

Пример проверяет работу вытесняющего планировщика задач. Он демонстрирует, как система переключается между задачами и управляет их выполнением. В рамках примера добавляются две задачи, каждая из которых выполняет следующие действия: увеличивает значение глобального счетчика на 1, выводит его текущее значение и добавляет задержку (имитация выполнения длительных операций). Если значение счетчика превышает 20, задача завершается.

3.7. Написание примера на C

В примерах на Rust библиотека `cortex-m-rt` автоматически генерирует код инициализации, таблицу векторов прерываний и выполняет настройку аппаратного обеспечения. Однако при разработке примера на языке C все эти задачи необходимо реализовать самостоятельно. Для этого требуется создать несколько ключевых файлов: файл конфигурации линкера, файл инициализации системы и основной код приложения. Дерево объектов файловой системы примера:

```
c-example/  
├── ld/  
│   └── stm32f4.ld  
├── src/  
│   ├── startup.c  
│   └── main.c  
└── Makefile
```

3.7.1. Файл `stm32f4.ld`

Файл линкера (`.ld`) определяет размещение секций памяти в адресном пространстве микроконтроллера. Он играет ключевую роль в процессе компиляции и связывания программы, указывая, где должны находиться исполняемый код, данные и другие сегменты программы.

Данный файл создан на основе соответствующего файла из библиотеки `cortex-m-rt`, что обеспечивает корректное размещение всех необходимых секций.

3.7.2. Файл `startup.c`

Файл `startup.c` содержит код инициализации системы, который выполняется сразу после сброса микроконтроллера. Он отвечает за подготовку окружения для выполнения основного кода программы:

- Обработчик сброса:
 - копирует инициализированные данные (секция `.data`) из FLASH в RAM;
 - обнуляет неинициализированные данные (секция `.bss`);
 - с помощью ассемблерной вставки включает поддержку FPU;
 - вызывает функцию `main()` для запуска пользовательского кода.
- Вектор прерываний:
 - определяет таблицу векторов прерываний, которая содержит указатели на обработчики соответствующих прерываний.

Файл `startup.c` выполняет роль "точки входа" программы, обеспечивая минимальную инициализацию системы перед вызовом пользовательского кода.

Листинг 1: Файл openocd.cfg.

```
source [find interface/stlink.cfg]
source [find target/stm32f4x.cfg]
```

Листинг 2: Команды для запуска gdb-multiarch.

```
openocd -f openocd.cfg
```

3.7.3. Файл main.c

Файл `main.c` содержит пользовательский код примера, который проверяет базовую работу системы: инициализацию системы, добавление задачи в невытесняющий планировщик и его запуск. Суть задачи заключается в следующем: глобальная переменная увеличивается на каждой итерации цикла задачи и если значение этого счетчика достигает 50, задача завершается.

3.8. Запуск примеров на плате

Для тестирования примеров использовались инструменты, рассмотренные в обзоре: OpenOCD и gdb-multiarch.

3.8.1. Конфигурация OpenOCD

OpenOCD запускается с конфигурацией из файла `openocd.cfg` (Листинг 1). Этот файл содержит настройки для интерфейса отладчика (ST-LINK) и целевого устройства (STM32F4). Команда для запуска представлена в листинге 2.

При запуске OpenOCD с этим конфигурационным файлом происходит подключение к микроконтроллеру через ST-LINK, что позволяет выполнять операции загрузки программы, сброса устройства и отладки.

Листинг 3: Команды для запуска OpenOCD.

```
gdb-multiarch -x openocd.gdb <путь-к-исполняемому-файлу>.elf
```

Листинг 4: Файл openocd.gdb.

```
target extended-remote :3333
monitor reset halt
load
monitor arm semihosting enable
continue
```

3.8.2. Скрипт GDB

Для взаимодействия с OpenOCD используется gdb-multiarch (Листинг 3). Файл openocd.gdb (Листинг 4) содержит команды для автоматизации процесса отладки через GDB. Он устанавливает соединение между GDB и OpenOCD, отправляет команду через OpenOCD для сброса микроконтроллера и перевода его в состояние остановки, загружает скомпилированный исполняемый файл в память микроконтроллера, включает режим semihosting, позволяющий микроконтроллеру взаимодействовать с хост-компьютером через отладочный интерфейс и запускает выполнение программы на микроконтроллере.

4. Тестирование и апробация

4.1. Тестирование

Поскольку реализация портирования включает либо прослойку между подключаемыми библиотеками, либо код, тесно связанный с аппаратной платформой, модульное тестирование не проводилось. Работоспособность демонстрационных примеров проверялась на физической плате. Кроме того, в рамках CI были реализованы интеграционные тесты с использованием эмулятора QEMU: демонстрационные примеры запускаются с помощью `qemu-system-arm`, после чего с помощью утилиты `grep` проверяется соответствие вывода ожидаемому результату. Однако из-за того, что QEMU не может полностью эмулировать работу таймеров для STM32F4, примеры `scheduler` и `timer` не тестируются.

4.2. Апробация

Апробация разработанного решения проводилась совместно с консультантом, являющимся также представителем компании. В ходе оценки были рассмотрены и одобрены такие аспекты, как его работоспособность и степень соответствия реализованной функциональности предъявленным требованиям и ожиданиям.

5. Заключение

В ходе выполнения данной работы были достигнуты все поставленные задачи:

- проведен обзор основных компонентов операционной системы и доступных инструментов для разработки;
- реализована поддержка базовых функций операционной системы;
- разработан набор демонстрационных примеров на языках программирования Rust и C.

Код работы является закрытым.

Список литературы

- [1] ARM Microcontrollers Market Size, Share, Competitive Landscape and Trend Analysis Report, by Product, by Application: Global Opportunity Analysis and Industry Forecast, 2024-2032. — Access mode: <https://www.alliedmarketresearch.com/arm-microcontrollers-market-A64747> (online; accessed: 13 июня 2025 г.).
- [2] Cortex-M4 Programming manual. — Access mode: <https://www.cse.chalmers.se/edu/resources/mop/documents/STM32F-ProgrammingManual.pdf> (online; accessed: 13 июня 2025 г.).
- [3] Low level access to Cortex-M processors. — Access mode: https://docs.rs/cortex-m/0.7.7/cortex_m/ (online; accessed: 13 июня 2025 г.).
- [4] Multi device hardware abstraction on top of the peripheral access API for the STMicro STM32F4 series microcontrollers. — Access mode: https://docs.rs/stm32f4xx-hal/latest/stm32f4xx_hal/ (online; accessed: 13 июня 2025 г.).
- [5] Open On-Chip Debugger. — Access mode: <https://openocd.org/pages/about.html> (online; accessed: 13 июня 2025 г.).
- [6] Startup code and minimal runtime for Cortex-M microcontrollers. — Access mode: https://docs.rs/cortex-m-rt/0.7.5/cortex_m_rt/ (online; accessed: 13 июня 2025 г.).
- [7] A heap allocator for embedded systems. — Access mode: <https://github.com/rust-embedded/embedded-alloc/tree/master> (online; accessed: 13 июня 2025 г.).
- [8] Отчёт по осенней учебной практике. — Access mode: https://se.math.spbu.ru/thesis/texts/Rodionov_Maksim_Aleksandrovich_Autumn_practice_2nd_year_2024_text.pdf (online; accessed: 13 июня 2025 г.).