Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б11-мм

# C-совместимость операционной системы, реализованной на Rust (RISC-V)

***Орешин Константин Денисович***

Отчёт по учебной практике
в форме «Решение»

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

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

Санкт-Петербург
2025

# Оглавление

# Введение

Martos[1] — операционная система реального времени, написанная на языке программирования Rust. Она позволяет запускать программы на различных процессорных архитектурах, таких как MIPS64, RISC-V, Xtensa. Также Martos поддерживает: диспетчер задач на основе Round-robin алгоритма, аллокацию памяти, взаимодействие с аппаратными таймерами.

Принцип работы Martos похож на концепцию операционной системы FreeRTOS [3] — в ней нет таких составляющих как ядро или процесс, только такие сущности как tasks. Исполняемая программа связывается с этой ОС, порождается директория сборки, которая уже загружается на плату или SoC[2] с соответствующей архитектурой. Затем пользовательское приложение вызывает функции Martos, таким образом взаимодействуя с ней.

Прикладные программы, написанные на Rust, позволяют вызывать функции ОС напрямую. Однако не все пишут на Rust, язык C более распространён среди программистов. [10] Поэтому, чтобы Martos был удобен большему числу людей, хотелось бы сделать эту ОС C-совместимой, то есть такой, которая бы смогла запускать программы и на C.

Для этого нужна статическая библиотека Martos C, с которой уже будет связываться исполняемая программа, дальнейший результат загружаться на плату и исполняться.

На текущем этапе реализована сборка программы на Rust под SoC ESP32C6 и связанную с ней архитектуру RISC-V. Необходимо расширить данный пример — собрать и связать с Martos программу на C, чтобы она смогла запуститься на этой же архитектуре.

[1] https://github.com/IvanArkhipov1999/Martos

[2] System-on-a-Chip

# 1. Постановка задачи

Целью работы является реализация примера программы на C, его сборка и связывание с Martos. Для её выполнения были поставлены следующие задачи:

1. провести обзор существующих подобных операционных систем и их подходов к решению данной проблемы;
2. написать сам пример, использующий функции Martos, и собрать из него исполняемый файл;
3. протестировать работу файла на плате ESP32C6 с архитектурой RISC-V;

# 2. Обзор

В данном разделе перечислены существующие операционные системы, написанные на Rust. В каждой из них присутствует возможность исполнения кода на C, нужно выбрать лучший подход. Перед обзором следует выделить некоторые требования, на основании которых будет приниматься решение об использовании конкретного инструмента:

1. открытый исходный код — для изучения конкретного подхода потребуется исходный код ОС;

2. поддержка RISC-V — метод может не работать на некоторых архитектурах, нужно убедиться, что у него присутствует поддержка RISC-V;

3. встраиваемость — необходим легковесный подход, который бы подошёл для встраиваемых систем.

## 2.1. Tock

Tock [12] — встраиваемая операционная система с открытым исходным кодом, предназначенная для маломощных платформ с небольшим количеством памяти, основанных на ARM Cortex-M процессорах или RISC-V архитектуре. Она предоставляет возможность работать с несколькими конкурентными приложениями.

Tock позволяет запускать программы, написанные на C и C++. Для этого используется libtock-c библиотека. Она состоит из трёх главных компонентов: crt0.c файл, запускающийся перед пользовательским приложением, конфигурирует память для инициализации C runtime; интерфейсы для libc, позволяющие вызывать функции оттуда; драйверы для API системных вызовов, которые предоставляет ядро Tock.

Для компиляции C кода на Tock требуется система сборки Make. Разработчику в директории проекта нужно прописать стандартный Makefile, в котором указать обязательные параметры сборки: пере-

менную `TOCK_USERLAND_BASE_DIR` — путь до Tock userland[3] и include AppMakefile.mk — специальный Makefile, написанный заранее разработчиками Tock и содержащий главные особенности сборки. В качестве дополнительных опций можно указать `C_SRCS`, `CXX_SRCS`, `AS_SRCS` — списки файлов C, C++, assembly для компиляции соответственно.

Система позволяет вручную выделить необходимые ресурсы и конкретизировать параметры компиляции. С помощью специальной переменной `STACK_SIZE` можно указать минимальный необходимый размер для стека, `APP_HEAP_SIZE` — размер для кучи, `KERNEL_HEAP_SIZE` — размер для всего приложения, а `PACKAGE_NAME` предоставляет возможность задать имя приложению.

AppMakefile.mk генерирует во время выполнения правила, необходимые для построения libtock-c приложения для нужной архитектуры и запускает их. Результат компиляции собирается в единый TAB[4] файл. AppMakefile также запускает другие файлы для сборки: Configuration.mk и Helpers.mk. Configuration.mk выставляет практически все параметры и флаги, требуемые для сборки libtock-c приложений. Helpers.mk прописывает вспомогательные функции и определения для использования их файлами выше.

## 2.2. OxidOS

OxidOS [7] — встраиваемая коммерческая операционная система. Нацелена на рынок автомобильной промышленности, поэтому основной её задачей является обеспечение гарантий безопасности. Это достигается запуском изолированных в памяти приложений, не имеющих доступ напрямую к аппаратному обеспечению. Также позволяет исполнять пользовательские программы на C или C++. Подходит для ECU[5], HSE[6], гипервизоров на архитектурах ARM и RISC-V.

Однако у ОС закрытый исходный код, оттого нельзя точно сказать,

[3]программы, запущенные в пользовательском пространстве
[4]TOCK APPLICATION BUNDLES
[5]Electronic Control Unit
[6]Hardware Security Engine

какими способами она реализует поддержку кода на C.

## 2.3. Theseus

Theseus [11] — операционная система с открытым кодом, находящаяся в активной разработке. Предназначается для лидирующих встраиваемых систем и сред обработки данных. Однако уже сейчас присутствует экспериментальная поддержка сборки программ на C, но только для архитектуры `x86_64`.

Theseus также предоставляет свою реализацию libc — tlibc. Она является статической библиотекой, с которой должно связываться приложение на C. Это происходит с помощью одного Makefile, в котором задаются специальные флаги и ограничения.

Theseus не поддерживает разделения на kernel-space и user-space. Все приложения в ней работают в привилегированном режиме, из-за этого возникает необходимость отключения красной зоны стека: любые данные, находящиеся в ней, могут быть перезаписаны другим процессом, для этого используется флаг `-mno-red-zone`. Применяется флаг `-pie` для вычисления адресов сущностей по смещению от адреса начала программы, а не по их абсолютному адресу.

После этапа компиляции загрузчик Theseus в объектном фале проходится по всем обращениям к секциям .data, .rodata и заменяет такие вхождения на соответствующие существующие Theseus-секции, в результате чего этого приложение будет использовать именно их в процессе работы.

## 2.4. Redox

Redox [8] — Unix-подобная микроядерная операционная система с открытым исходным кодом. Позиционируется как альтернатива Linux и BSD.

Redox использует свой вариант libc — relibc. [9] Это стандартная библиотека языка C, полностью переписанная на Rust. Она POSIX-совместимая, за счёт этого в Redox можно запускать пользовательский

Таблица 1: соответствие предложенных вариантов критериям

| | Open-source | RISC-V support | Embeddability |
|---|---|---|---|
| Tock | + | + | + |
| OxidOS | - | + | + |
| Theseus | + | - | + |
| Redox | + | - | - |

C код и использовать уже существующие программы, написанные на C, такие как Git или Python. У приложений также есть возможность делать системные вызовы, несмотря на то, что функции в kernel-space написаны на Rust.

Redox также поддерживает gcc13, позволяющий компилировать C код в бинарные файлы Redox, использующие обычный ELF-формат.

Однако такое решение не подходит для встраиваемых систем — relibc и gcc13 слишком тяжеловесны, будет проблематично портировать их на SoC или плату. Вдобавок, Redox поддерживает только архитектуры `x86_64`, AMD64, ARM64.

## 2.5. Вывод

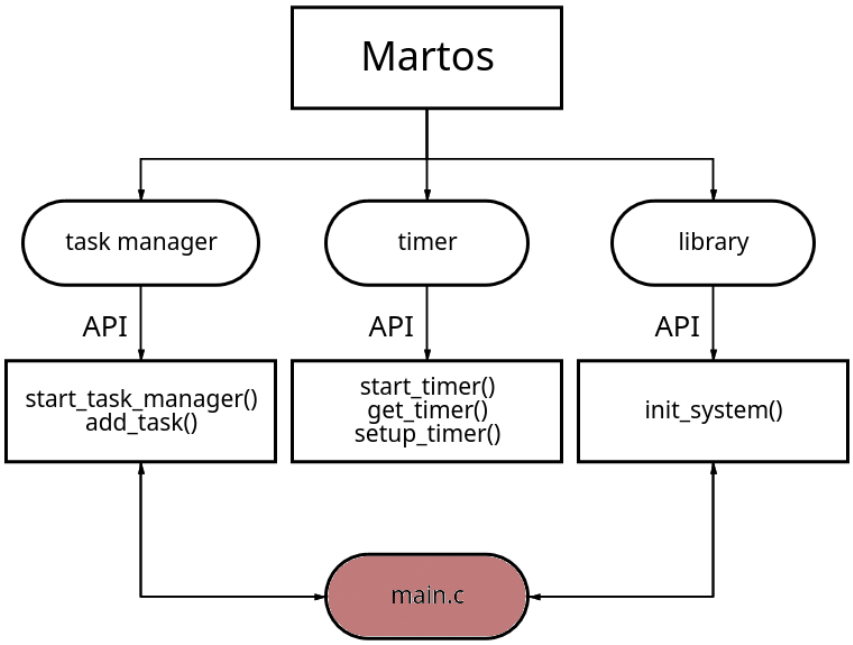
Таким образом, можно сделать вывод, что подход, используемый в операционной системе Tock, удовлетворяет всем необходимым требованиям. Метод связывания, который будет реализовываться в Martos, очень схож с идеей Tock: также используется статическая библиотека ОС, также связь с ней происходит через Makefile, результат работы годится для встраиваемых систем и архитектуры RISC-V.

# 3. Реализация

Пример на C должен взаимодействовать с Martos через его API. ОС и статическая библиотека предоставляют множество функций для взаимосвязи с ними: `start_timer()`, `set_reload_mode()`, `get_time()` и другие. Однако интерес представляют только 3: `init_system()`, `add_task()`, `start_task_manager()`.

Первая инициализирует систему, включая все её компоненты: кучу, таймер, сеть. Вторая принимает три аргумента: setup, loop и condition функции. Одна выполняется при старте, вторая — пока верно условие третьего предиката. `Start_task_manager()` запускает менеджер задач.

Перед написанием примера стоит собрать статическую библиотеку Martos, с которой он будет связываться. Для этого нужно действовать в соответствии с README.md в директории c-library/risc-v-esp32-c6/, после чего в target/riscv32imac-unknown-none-elf/release/ будет лежать собранная библиотека. В examples/c-examples необходимо создать директорию risc-v-esp32c6/, где будет располагаться сам пример.



```
void setup_fn() {
}
void loop_fn() {
    counter++;
}
bool stop_condition_fn() {
    if (counter == 50) {
        return true;
    }
    return false;
}

int main( void ) {
    init_system();
    add_task(setup_fn, loop_fn, stop_condition_fn);
    start_task_manager();
    return 0;
}
void __attribute__( ( noreturn ) ) call_start_cpu0() {
    main();
    while( 1 ) {}
}
```

## 3.1. Написание примера

В качестве примера работы Martos была выбрана простая концепция программы, производящей инкремент переменной 50 раз и прекращающей работу после этого. Она использует ключевые функции ОС: инициализация системы, добавление задачи и последующий запуск менеджера. Всё остальное взаимодействие с Martos, включая использования таймера, так или иначе будет происходить через эти компоненты и базироваться на них. Из этого следует дальнейшая реализация примера. В `main()` совершается вызов вышеперечисленных функций с передачей в `add_task()` нужных составляющих для выполнения поставленной задачи.

Однако точкой входа в программу будет далеко не `main()`, а стан-

**Листинг 1: Спецификация архитектуры и ABI, отключение строгого выравнивания, отключение стандартных файлов и библиотек запуска, включение специальных расширений GNU**

```
CFLAGS = -march=rv32imac -mabi=ilp32
CFLAGS += -fno-strict-aliasing -nostdlib -std=gnu11
```

**Листинг 2: Связывание со статической библиотекой, создание промежуточного .elf.map файла**

```
LDFLAGS += -lrisc_v_esp32c6_static_lib
-Wl,-Map=$@.map -Wl,--cref -lgcc
```

дартная для ESP-IDF приложений `call_start_cpu0()`. [1] Она инициализирует базовое окружение C Runtime, производит первичную настройку внутреннего аппаратного обеспечения SoC, а после уже вызывает саму `main()`.

## 3.2. Написание Makefile

Для повышения удобства и автоматизации процесса сборки примера был написан Makefile. Под архитектуру RISC-V компиляцию производится с помощью riscv32-esp-elf-gcc. Для него в CFLAGS прописываются флаги включения всех предупреждающих сообщений, отключения строгого выравнивания, добавления отладочной информации, спецификации rv32imac архитектуры, ilp32 ABI и другие. (Листинг 1)

В переменную LDFLAGS (Листинг 2) записываются флаги, которые отключают использование стандартных для системы файлов запуска и библиотек, производят связывание с собранной статической библиотекой Martos, генерируют таблицу перекрёстных ссылок и перенаправляют её в main.elf.map, используют опцию gc-sections, которая вместе с флагом fdata-sections убирает весь неиспользуемый код.

Наконец, прописав все зависимости, производится в два этапа ком-

**Листинг 3: Указание точки входа**

```
ENTRY(call_start_cpu0);
```

пиляция и связывание.

## 3.3. Написание linker script

Следующим этапом будет написание linker script [4], который выполняет 3 основные функции:

1. указывает точку входа в программу;
2. раскладывает секции по адресам в исполняемом файле;
3. помогает разрешить неопределённые символы.

За основу был взят стандартный скрипт для ESP32C6 [2] и видоизменён в соответствии с требованиями программы. Так как она загружается в оперативную память, то и затрагиваются только области HP-SRAM и LP-SRAM, остальные сегменты были удалены. Реализация скрипта хранится в файле esp32.ld в директории ld примера.

Выделяются сегменты по следующим адресам: [5]
0x4085e9d0 - 0x408629d0 –> `iram_seg`
0x408629d0 - 0x408699d0 –> `iram_loader_seg`
0x408699d0 - 0x4087a610 –> `dram_seg`
0x50000000 - 0x50003fff –> `lp_ram_seg`
В эти сегментах размещаются следующие секции: [6]

1. `iram_seg` содержит код инициализации .init, код завершения .fini, пользовательский код .text;
2. `iram_loader_seg` содержит код, который может быть использован программой после второй стадии загрузчика;

3. `dram_seg` содержит область неинициализированных данных .bss, инициализированных с правами на чтение и запись .data и с правами только на чтение .rodata;

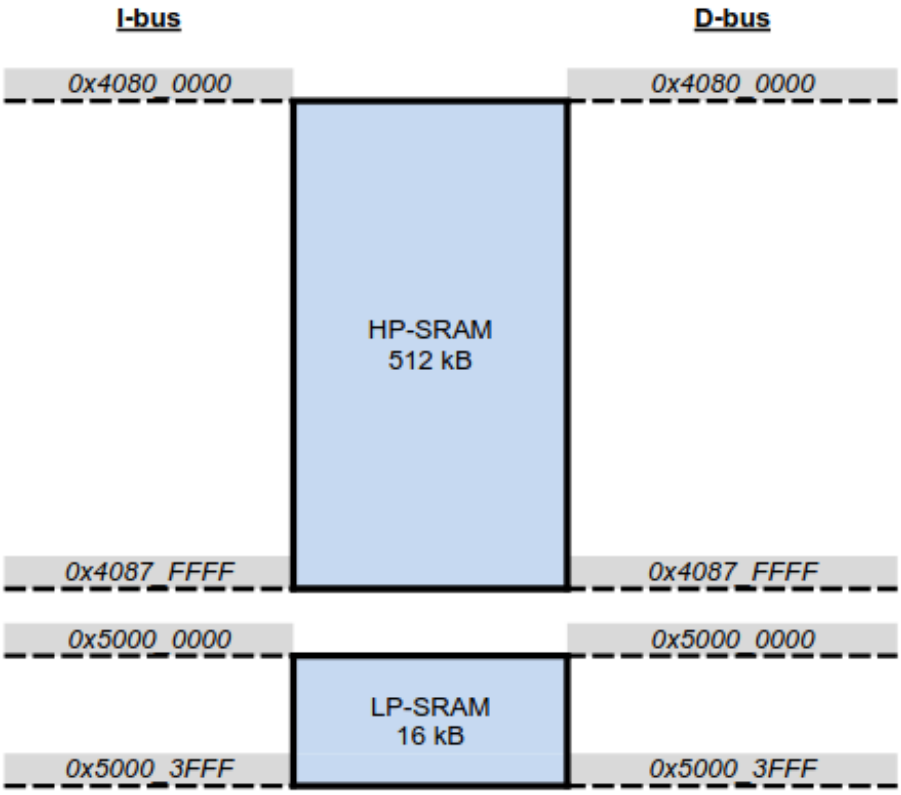
4. `lp_ram_seg` содержит код, который выполняется сразу после выхода платы из deep-sleep состояния. Она содержит такие секции, как .rtc.data, .rtc.text, .rtc.bss и подобные.

При попытках собрать программу без linker script неизбежно столкновение с проблемой неразрешимых имён: статическая библиотека, с которой происходит связывание, ссылается на символы, но не определяет их. Они включают в себя границы секций, такие как `_rtc_fast_start`, `_bss_start`, `_dram_end` и подобные. Linker script разрешает эту проблему и явно указывает адреса данных символов.

В Makefile через флаг -T необходимо прописать указание компоновщику заиспользовать его.

Затем происходит сборка примера в 2 этапа: компиляция и связывание. Результатом этой деятельности должен стать исполняемый файл main.elf, который впоследствии и будет загружаться на плату.

Рис. 1: Карта памяти ESP32C6 https://dl.espressif.com/public/esp32c6-mm.pdf




# 4. Запуск примера

## 4.1. Запуск в QEMU

Была предпринята попытка запуска приложения через эмулятор программного обеспечения QEMU, однако возникли проблемы и сложности со сборкой приложения. Это объясняется тем, что поддержка QEMU для ESP-приложений находится ещё в разработке и официально не задокументирована.

## 4.2. Запуск на плате

Для запуска примера на плате используется утилита esptool.py версии 4.7.0. В первую очередь необходимо отформатировать исполняемый файл в бинарный образ. Для этого указываются указывается тип платы, тип соединения, тактовая частота, размер. (Листинг 4)

В Листинге 5 представлены параметры записи бинарного образа на

**Листинг 4: Форматирование в бинарный образ**

```
    --chip esp32c6 --flash_freq "40m"
    --flash_size "4MB" elf2image
```

**Листинг 5: Запись на плату**

```
    --port /dev/ttyUSB0 --baud 115200 write_flash
```

плату. Для этого указываются порт, скорость передачи, адрес загрузчика.

В Листинге 6 прописаны параметры запуска корректно отрабатывающего примера на плате.

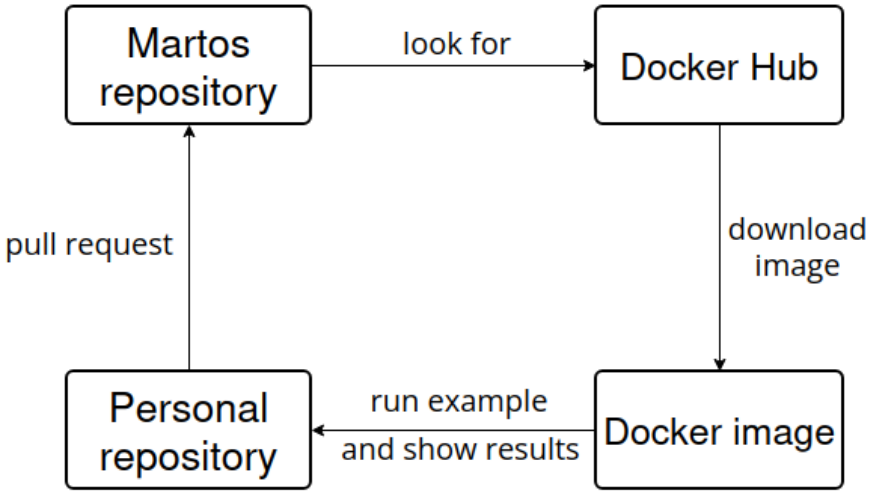
## 4.3. Запуск в CI

Также запуск примера был добавлен в CI репозитория Martos. При попытках внести изменения через запросы слияния через GitHub Actions из Docker Hub будет загружаться образ с предустановленным для запуска окружением ESP32C6.

По результатам последующего запуска примера в Docker контейнере, аналогичного с запуском его на плате, будет выявлено, не нарушают ли изменения, затронутые в pull request, целостности репозитория и готовы ли они к слиянию.

**Листинг 6: Запуск примера**

```
    --before default_reset --after hard_reset run
```




# Заключение

Был реализован пример программы на C, совместимой с многоагентной операционной системой для встраиваемых систем Martos

- проведен обзор существующих подходов к решению похожей проблемы;
- реализован пример работы программы на C, вызывающей функции Martos;
- сборка примера добавлена в CI репозитория с проектом.

Запросы на слияние с реализацией примера
https://github.com/IvanArkhipov1999/Martos/pull/76
https://github.com/IvanArkhipov1999/Martos/pull/84
Запрос на слияние с добавлением установки окружения из Docker Hub для последующего запуска примера в нём
https://github.com/IvanArkhipov1999/Martos/pull/78

# Список литературы


[1] Application Startup Flow. — Access mode: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-guides/startup.html (online; accessed: 28 ноября 2024 г.).

[2] Espressif Systems. — ESP32C6. Technical Reference Manual : 2024. — Access mode: https://www.espressif.com/sites/default/files/documentation/esp32-c6_technical_reference_manual_en.pdf.

[3] FreeRTOS. Real-time operating system for microcontrollers and small microprocessors. — Access mode: https://www.freertos.org/ (online; accessed: 28 ноября 2024 г.).

[4] GNU linker ld documentation. — Access mode: https://sourceware.org/binutils/docs/ld.html (online; accessed: 28 ноября 2024 г.).

[5] Memory map for ESP32C6. — Access mode: https://docs.mcuboot.com/readme-espressif.html#memory-map-organization-for-os-compatibility (online; accessed: 28 ноября 2024 г.).

[6] Memory types for ESP32C6. — Access mode: https://docs.espressif.com/projects/esp-idf/en/stable/esp32c6/api-guides/memory-types.html (online; accessed: 28 ноября 2024 г.).

[7] OxidOS. Rust-based secure ecosystem for safety critical automotive ECUs. — Access mode: https://oxidos.io/ (online; accessed: 28 октября 2024 г.).

[8] Redox. Unix-like general-purpose microkernel-based operating system written in Rust. — Access mode: https://www.redox-os.org/ (online; accessed: 28 октября 2024 г.).

[9] Relibc implementation. — Access mode: https://gitlab.redox-os.org/JCake/relibc (online; accessed: 28 ноября 2024 г.).

[10] TIOBE Index for December 2024. — Access mode: https://www.tiobe.com/tiobe-index/ (online; accessed: 30 декабря 2024 г.).

[11] Theseus. OS written from scratch in Rust. — Access mode: https://www.theseus-os.com/ (online; accessed: 28 октября 2024 г.).

[12] Tock. An embedded operating system. — Access mode: https://tockos.org/ (online; accessed: 28 октября 2024 г.).