The Embedded Rust Book (Russian)

Contents

1. Введение 5
    - 1.0.1. Для кого предназначен Embedded Rust 5
    - 1.0.2. Область применения 5
    - 1.0.3. Для кого эта книга 5
    - 1.0.4. Повторное использование материала 7
  - 1.1. Знакомство с оборудованием 7
    - 1.1.1. STM32F3DISCOVERY («F3») 8
  - 1.2. Среда `no_std` в Rust 8
    - 1.2.1. Hosted-среды 9
    - 1.2.2. Bare-metal среды 9
    - 1.2.3. Итог 9
    - 1.2.4. См. также 10
  - 1.3. Инструменты 10
    - 1.3.1. `cargo-generate` ИЛИ `git` 10
    - 1.3.2. `cargo-binutils` 11
    - 1.3.3. `qemu-system-arm` 11
  - 1.4. Инструменты для отладки встраиваемого Rust 11
    - 1.4.1. Обзор 11
    - 1.4.2. Программное обеспечение, управляющее инструментами отладки 11
    - 1.4.3. Отладчики 12
    - 1.4.4. Зонды 12
  - 1.5. Установка инструментов 13
    - 1.5.1. Linux 14
    - 1.5.2. macOS 16
    - 1.5.3. Windows 16
    - 1.5.4. Проверка установки 17
2. Начало работы 19
  - 2.1. QEMU 19
    - 2.1.1. Создание нестандартной программы на Rust 19
    - 2.1.2. Обзор программы 20
    - 2.1.3. Кросс-компиляция 21
    - 2.1.4. Инспекция 22
    - 2.1.5. Запуск 25
    - 2.1.6. Отладка 26
3. Оборудование 30
    - 3.0.1. Знакомство с вашим оборудованием 30
    - 3.0.2. Настройка 30
    - 3.0.3. Отладка 31
  - 3.1. Регистры, отображённые в память 35
    - 3.1.1. Крейт платы 36
    - 3.1.2. Крейт микроархитектуры 36
    - 3.1.3. Использование Peripheral Access Crate (PAC) 37
    - 3.1.4. Использование HAL-крейта 39
  - 3.2. Semihosting 40
  - 3.3. Паника 43
    - 3.3.1. Пример 44
  - 3.4. Исключения 45

3.4.1. Полный пример ... 46
3.4.2. Обработчик исключений по умолчанию ... 47
3.4.3. Обработчик Hard Fault ... 48
3.5. Прерывания ... 50
4. Периферия ... 52
4.1. Что такое периферия? ... 52
4.2. Линейное адресное пространство и реальная память ... 52
4.3. Периферия, отображённая в память ... 53
4.4. Первая попытка ... 54
4.4.1. Регистры ... 54
4.4.2. Подход C ... 54
4.4.3. Летучие (volatile) обращения ... 54
4.4.4. Удобная обёртка на Rust ... 55
4.5. Изменяемое глобальное состояние ... 57
4.5.1. Какими должны быть наши правила? ... 57
4.5.2. Анализатор заимствований (borrow checker) ... 57
4.6. Синглтоны ... 57
4.6.1. Почему нельзя просто использовать глобальные переменные? ... 57
4.6.2. Как это сделать в Rust? ... 58
4.6.3. Поддержка в существующих библиотеках ... 58
4.6.4. Но зачем? ... 59
4.6.5. Относитесь к аппаратуре как к данным ... 60
5. Статические гарантии ... 61
5.1. Программирование на typestate ... 61
5.1.1. Строгие типы ... 62
5.2. Периферия как конечные автоматы ... 62
5.2.1. Представление в аппаратуре ... 63
5.3. Проектные контракты ... 64
5.3.1. Состояния в типах (type states) ... 67
5.3.2. Функциональная безопасность на этапе компиляции ... 70
5.4. Абстракции с нулевой стоимостью ... 70
5.4.1. Типы нулевого размера ... 70
5.4.2. Вложенность ... 71
6. Переносимость ... 72
6.0.1. Что такое embedded-hal? ... 72
6.0.2. Пользователи embedded-hal ... 73
7. Конкурентность ... 75
7.0.1. Без конкурентности ... 75
7.0.2. Глобальные изменяемые данные ... 75
7.0.3. Критические секции ... 76
7.0.4. Атомарный доступ ... 77
7.0.5. Абстракции, Send и Sync ... 78
7.0.6. Мьютексы ... 80
7.0.7. Совместное использование периферии ... 82
7.0.8. RTIC ... 85
7.0.9. Embassy ... 85
7.0.10. Операционные системы реального времени ... 86
7.0.11. Несколько ядер ... 86
8. Коллекции ... 87

8.0.1. Использование `alloc` ... 87
8.0.2. Использование `heapless` ... 89
8.0.3. Компромиссы ... 90
9. Паттерны проектирования ... 92
9.1. Паттерны проектирования HAL ... 92
9.1.1. Чек-лист паттернов проектирования HAL ... 92
9.1.2. Именование ... 92
9.1.3. Совместимость ... 92
9.1.4. Предсказуемость ... 93
9.1.5. Рекомендации для интерфейсов GPIO ... 94
10. Советы для разработчиков встраиваемого C ... 98
10.0.1. Препроцессор ... 98
10.0.2. Система сборки ... 100
10.0.3. Итераторы вместо доступа по индексу ... 100
10.0.4. Ссылки и указатели ... 101
10.0.5. Летучий (volatile) доступ ... 101
10.0.6. Упакованные и выровненные типы ... 102
10.0.7. Другие ресурсы ... 105
11. Взаимодействие ... 106
11.0.1. Взаимодействие с другими системами сборки ... 106
11.0.2. Взаимодействие с RTOS ... 106
11.1. Немного C в вашем Rust ... 106
11.1.1. Определение интерфейса ... 107
11.1.2. Сборка вашего кода на C/C++ ... 108
11.2. Немного Rust в вашем C ... 110
11.2.1. Настройка проекта ... 110
11.2.2. Построение API на C ... 110
11.2.3. Компоновка и общий контекст проекта ... 111
12. Разные темы ... 112
12.1. Оптимизации: компромисс между скоростью и размером ... 112
12.1.1. Без оптимизаций ... 112
12.1.2. Оптимизация по скорости ... 113
12.1.3. Оптимизация по размеру ... 114
12.2. Математические вычисления в `#[no_std]` ... 115
A. Приложение A: Глоссарий ... 117

1. Введение

Добро пожаловать в книгу «Встраиваемый Rust» (The Embedded Rust Book) — вводное руководство по использованию языка программирования Rust во встраиваемых системах «bare metal», например на микроконтроллерах.

1.0.1. Для кого предназначен Embedded Rust

Embedded Rust предназначен для всех, кто хочет заниматься встраиваемым программированием и при этом пользоваться высокоуровневыми концепциями и гарантиями безопасности, которые даёт язык Rust. (См. также Для кого предназначен Rust)

1.0.2. Область применения

Цели этой книги:

- Помочь разработчикам быстро освоить разработку на embedded Rust, то есть показать, как настроить среду разработки.
- Поделиться *актуальными* лучшими практиками использования Rust во встраиваемой разработке, то есть показать, как лучше использовать возможности языка, чтобы писать более корректное встраиваемое ПО.
- В некоторых случаях служить сборником рецептов, например: как смешать C и Rust в одном проекте?

Эта книга старается быть максимально общей, но чтобы облегчить жизнь и читателям, и авторам, все примеры в ней используют архитектуру ARM Cortex-M. При этом книга не предполагает, что читатель знаком с этой архитектурой, и там, где это нужно, объясняет её особенности.

1.0.3. Для кого эта книга

Книга рассчитана на тех, у кого есть либо опыт во встраиваемых системах, либо опыт работы с Rust. Однако мы уверены, что любой, кому интересно встраиваемое программирование на Rust, сможет извлечь из неё пользу. Если у вас нет предварительных знаний, мы советуем прочитать раздел «Предположения и предварительные требования» и восполнить недостающие знания, чтобы лучше усвоить материал книги. В разделе «Другие ресурсы» вы найдёте материалы по темам, которые могут вам понадобиться.

1.0.3.1. Предположения и предварительные требования

- Вы уверенно пользуетесь языком программирования Rust, писали, запускали и отлаживали приложения на Rust в настольной среде. Также вы должны быть знакомы с идиомами издания 2018 года, так как эта книга рассчитана на Rust 2018.
- Вы уверенно разрабатываете и отлаживаете встраиваемые системы на другом языке, например C, C++ или Ada, и знакомы с такими понятиями, как:
  - кросс-компиляция;
  - периферия, отображаемая в память;
  - прерывания;
  - распространённые интерфейсы, например I2C, SPI, Serial и др.

1.0.3.2. Другие ресурсы

Если вы не знакомы с чем-либо из перечисленного выше или хотите узнать больше о какой-либо теме, упомянутой в этой книге, вам могут пригодиться следующие ресурсы.

Тема	Ресурс	Описание
Rust	Rust Book	Если вы ещё не освоились с Rust, настоятельно рекомендуем прочитать эту книгу.
Rust, Embedded	Discovery Book	Если вы никогда не занимались встраиваемым программированием, эта книга может стать лучшим стартом
Rust, Embedded	Embedded Rust Bookshelf	Здесь вы найдёте другие ресурсы, подготовленные рабочей группой Rust Embedded.
Rust, Embedded	Embedonomicon	Подробности и тонкости встраиваемого программирования на Rust.
Rust, Embedded	embedded FAQ	Часто задаваемые вопросы о Rust в контексте встраиваемых систем.
Rust, Embedded	Comprehensive Rust 🦀: Bare Metal	Учебные материалы для четырёхдневного курса по разработке на Rust для bare metal
Прерывания	Interrupt	-
Ввод-вывод и периферия, отображённые в память	Memory-mapped I/O	-
SPI, UART, RS232, USB, I2C, TTL	Вопрос на Stack Exchange о SPI, UART и других интерфейсах	-

1.0.3.3. Переводы

Эта книга переведена добровольцами. Если вы хотите, чтобы ваш перевод был указан здесь, откройте PR с его добавлением.

- Японский (репозиторий)
- Китайский (репозиторий)

1.0.3.4. Как пользоваться этой книгой

Эта книга в целом рассчитана на чтение от начала до конца. Более поздние главы опираются на понятия из предыдущих, а ранние главы могут не раскрывать тему полностью и возвращаются к ней в более поздних главах.

В большинстве примеров этой книги используется отладочная плата STM32F3DISCOVERY от STMicroelectronics. Эта плата основана на архитектуре ARM Cortex-M. Базовая функциональность у большинства процессоров на этой архитектуре одна и та же, однако периферия и другие детали реализации микроконтроллеров отличаются у разных производителей, а часто даже между семействами микроконтроллеров одного производителя.

Поэтому мы советуем приобрести плату STM32F3DISCOVERY, чтобы повторять примеры из книги.

1.0.3.5. Участие в работе над книгой

Работа над книгой ведётся в этом репозитории и в основном выполняется командой resources.

Если вам трудно следовать инструкциям из книги или какой-либо раздел кажется недостаточно понятным или сложным для восприятия, это ошибка, и о ней следует сообщить в трекере задач этой книги.

Pull request'ы с исправлениями опечаток и добавлением нового материала очень приветствуются!

1.0.4. Повторное использование материала

Эта книга распространяется по следующим лицензиям:

- Примеры кода и самостоятельные Cargo-проекты в этой книге распространяются по условиям лицензий MIT и Apache License v2.0.
- Написанный текст, изображения и диаграммы в этой книге распространяются по лицензии Creative Commons CC-BY-SA v4.0.

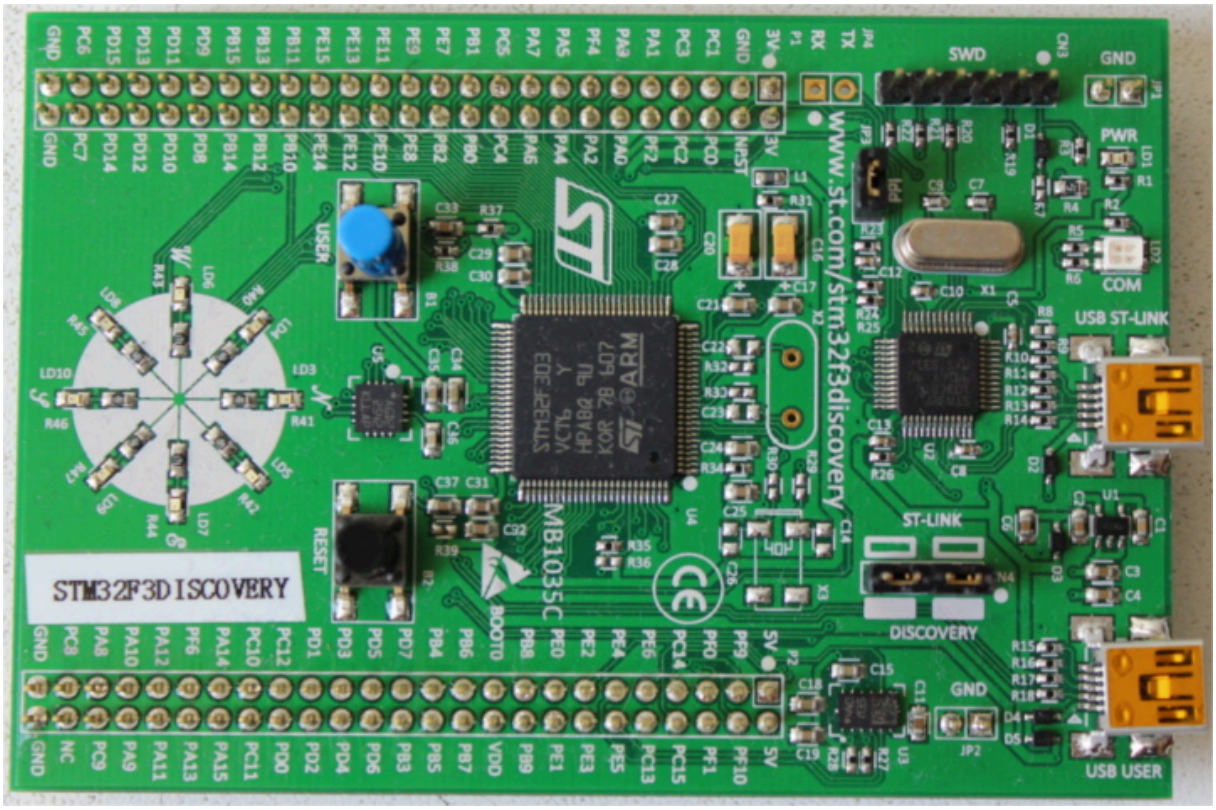
TL;DR: если вы хотите использовать наш текст или изображения в своей работе, вам нужно:

- Указать соответствующее авторство (например, упомянуть эту книгу на слайде и дать ссылку на нужную страницу)
- Привести ссылку на лицензию CC-BY-SA v4.0
- Указать, если вы каким-либо образом изменили материал, и сделать свои изменения доступными по той же лицензии

Также, пожалуйста, сообщите нам, если эта книга оказалась для вас полезной!

1.1. Знакомство с оборудованием

Давайте познакомимся с оборудованием, с которым мы будем работать.



1.1.1. STM32F3DISCOVERY («F3»)

Что есть на этой плате?

- Микроконтроллер STM32F303VCT6. Этот микроконтроллер содержит:
  - Одноядерный процессор ARM Cortex-M4F с аппаратной поддержкой операций с плавающей точкой одинарной точности и максимальной тактовой частотой 72 МГц.
  - 256 КиБ флеш-памяти. (1 КиБ = 1024 байта)
  - 48 КиБ ОЗУ.
  - Множество встроенной периферии, например таймеры, I2C, SPI и USART.
  - Выводы GPIO (универсальный ввод-вывод) и другие типы выводов, доступные через два ряда разъёмов вдоль платы.
  - Интерфейс Mini-USB, доступный через порт с надписью «USB USER».
- Акселерометр в составе микросхемы LSM303DLHC.
- Магнитометр в составе микросхемы LSM303DLHC.
- Гироскоп в составе микросхемы L3GD20.
- 8 пользовательских светодиодов, расположенных в форме компаса.
- Второй микроконтроллер: STM32F103. Он входит в состав встроенного программатора/отладчика и подключён к порту Mini-USB с надписью «USB ST-LINK».

Более подробный список возможностей и характеристик платы можно найти на сайте STMicroelectronics.

Важное предупреждение: будьте осторожны, если хотите подавать на плату внешние сигналы. Номинальное напряжение на выводах микроконтроллера STM32F303VCT6 составляет 3,3 В. Подробнее см. раздел 6.2 «Absolute maximum ratings» в документации.

1.2. Среда `no_std` в Rust

Термин «встраиваемое программирование» охватывает очень разные классы программ: от программирования 8-битных микроконтроллеров (например, ST72325xx) с несколькими

килобайтами ОЗУ и ПЗУ до систем вроде Raspberry Pi (Modell B 3+), у которой есть 32/64-битный четырёхъядерный Cortex-A53 на 1,4 ГГц и 1 ГБ ОЗУ. Ограничения, которые действуют при написании кода, зависят от типа целевой платформы и сценария использования.

Есть два общих класса встраиваемого программирования:

1.2.1. Hosted-среды

Такие среды близки к обычному ПК. Это значит, что вам предоставляется системный интерфейс например, POSIX, который даёт примитивы для взаимодействия с различными системами: файловыми системами, сетью, управлением памятью, потоками и т. д. Стандартные библиотеки, в свою очередь, обычно опираются на эти примитивы для реализации своей функциональности. Также у вас может быть какой-то sysroot и ограничения на использование ОЗУ и ПЗУ, а возможно, и специфическое оборудование или входы/выходы. В целом это ощущается как разработка для специализированного ПК.

1.2.2. Bare-metal среды

В bare-metal среде перед запуском вашей программы не загружается никакой код. Без программного обеспечения, которое предоставляет ОС, мы не можем загрузить стандартную библиотеку. Вместо этого программа вместе с крейтами, которые она использует, может работать только с аппаратным обеспечением (bare metal). Чтобы компилятор Rust не подключал стандартную библиотеку, используйте `no_std`. Платформонезависимые части стандартной библиотеки доступны через libcore. libcore также исключает то, что не всегда желательно во встраиваемой среде. Одна из таких вещей — аллокатор памяти для динамического выделения памяти. Если вам нужны он или любая другая функциональность, часто существуют крейты, которые её предоставляют.

1.2.2.1. Рантайм libstd

Как уже говорилось, использование libstd требует интеграции с какой-то системой. Причина не только в том, что libstd предоставляет общий способ доступа к абстракциям ОС, но и в том, что она включает рантайм. Этот рантайм, среди прочего, настраивает защиту от переполнения стека, обрабатывает аргументы командной строки и запускает главный поток до вызова функции `main` вашей программы. Этот рантайм также недоступен в среде `no_std`.

1.2.3. Итог

`#![no_std]` — это атрибут уровня крейта, который указывает, что крейт будет связываться с core-крейтом, а не со std-крейтом. Крейт libcore, в свою очередь, является платформонезависимым подмножеством std, которое не делает никаких предположений о системе, в которой будет работать программа. Поэтому он предоставляет API для примитивов языка, таких как числа с плавающей точкой, строки и срезы, а также API, которые раскрывают возможности процессора, например атомарные операции и SIMD-инструкции. Однако в нём нет API для всего, что связано с интеграцией с платформой. Из-за этих свойств код с `no_std` и libcore можно использовать для любого вида начального (stage 0) кода, например загрузчиков, прошивок или ядер.

1.2.3.1. Обзор

Возможность	no_std	std
heap (динамическая память)	*	✓
коллекции (Vec, BTreeMap и т. д.)	**	✓
защита от переполнения стека	✘	✓
инициализирующий код выполняется до main	✘	✓
libstd доступна	✘	✓
libcore доступна	✓	✓
разработка прошивки, ядра или загрузчика	✓	✘

* Только если вы используете крейт `alloc` и подходящий аллокатор, например alloc-cortex-m.

** Только если вы используете крейт `collections` и настраиваете глобальный аллокатор по умолчанию.

** HashMap и HashSet недоступны из-за отсутствия безопасного генератора случайных чисел.

1.2.4. См. также

- RFC-1184

1.3. Инструменты

Для работы с микроконтроллерами нужно несколько разных инструментов: мы будем работать с архитектурой, отличной от архитектуры вашего ноутбука, а запускать и отлаживать программы придётся на *удалённом* устройстве.

Мы будем использовать все перечисленные ниже инструменты. Если минимальная версия не указана, подойдёт любая свежая версия, но мы перечислили версии, которые тестировали.

- Rust 1.31, 1.31-beta или более новый тулчейн с поддержкой компиляции для ARM Cortex-M.
- `cargo-binutils` ~0.1.4
- `qemu-system-arm`. Протестированные версии: 3.0.0
- OpenOCD >=0.8. Протестированные версии: v0.9.0 и v0.10.0
- GDB с поддержкой ARM. Настоятельно рекомендуется версия 7.12 или новее. Протестированные версии: 7.10, 7.11, 7.12 и 8.1
- `cargo-generate` или `git`. Эти инструменты необязательны, но они облегчат работу с книгой.

Ниже объясняется, зачем нам нужны эти инструменты. Инструкции по установке приведены на следующей странице.

1.3.1. `cargo-generate` ИЛИ `git`

Bare-metal программы — это нестандартные программы на Rust (`no_std`), которые требуют некоторой настройки процесса компоновки, чтобы правильно задать раскладку памяти программы. Для этого нужны дополнительные файлы (например, скрипты компоновщика) и настройки (например, флаги компоновщика). Мы упаковали их в шаблон, чтобы вам нужно было заполнить только недостающую информацию (например, название проекта и характеристики целевого оборудования).

Наш шаблон совместим с `cargo-generate` — подкомандой Cargo для создания новых проектов Cargo из шаблонов. Также вы можете скачать шаблон с помощью `git`, `curl`, `wget` или веб-браузера.

1.3.2. `cargo-binutils`

`cargo-binutils` — это набор подкоманд Cargo, которые упрощают использование инструментов LLVM, поставляемых вместе с тулчейном Rust. Эти инструменты включают версии `objdump`, `nm` и `size` от LLVM и используются для анализа бинарных файлов.

Преимущество этих инструментов перед GNU binutils в том, что (а) установка инструментов LLVM выполняется одной и той же командой (`rustup component add llvm-tools`) независимо от вашей ОС и (б) такие инструменты, как `objdump`, поддерживают все архитектуры, которые поддерживает `rustc`, — от ARM до x86_64, так как все они используют один и тот же бэкенд LLVM.

1.3.3. `qemu-system-arm`

QEMU — это эмулятор. В данном случае мы используем вариант, который может полностью эмулировать системы ARM. Мы используем QEMU, чтобы запускать встраиваемые программы на хосте. Благодаря этому вы сможете пройти некоторые части этой книги, даже если у вас нет оборудования под рукой!

1.4. Инструменты для отладки встраиваемого Rust

1.4.1. Обзор

Отладка встраиваемых систем на Rust требует специализированных инструментов: программного обеспечения для управления процессом отладки, отладчиков для просмотра и контроля выполнения программы и аппаратных зондов, которые обеспечивают взаимодействие между хостом и встраиваемым устройством. В этом разделе описаны основные программные инструменты, такие как Probe-rs и OpenOCD, которые упрощают и поддерживают процесс отладки, а также популярные отладчики, такие как GDB и расширение Probe-rs для Visual Studio Code. Кроме того, рассмотрены ключевые аппаратные зонды — Rusty-probe, ST-Link, J-Link и MCU-Link, — которые необходимы для эффективной отладки и программирования встраиваемых устройств.

1.4.2. Программное обеспечение, управляющее инструментами отладки

1.4.2.1. Probe-rs

Probe-rs — современное программное обеспечение, ориентированное на Rust, которое предназначено для работы с отладчиками во встраиваемых системах. В отличие от OpenOCD, Probe-rs создан с прицелом на простоту и стремится уменьшить объём конфигурации, который часто требуется в других решениях для отладки. Он поддерживает различные зонды и целевые платформы и предоставляет высокоуровневый интерфейс для работы с встраиваемым оборудованием. Probe-rs интегрируется непосредственно с инструментами Rust, а через своё расширение — с Visual Studio Code, что позволяет разработчикам упростить процесс отладки.

1.4.2.2. OpenOCD (Open On-Chip Debugger)

OpenOCD — программный инструмент с открытым исходным кодом для отладки, тестирования и программирования встраиваемых систем. Он обеспечивает интерфейс между хост-системой и встраиваемым оборудованием и поддерживает различные транспортные уровни, например JTAG и SWD (Serial Wire Debug). OpenOCD работает вместе с GDB, который является отладчиком. OpenOCD широко поддерживается, имеет обширную документацию и большое сообщество, однако может требовать сложной конфигурации, особенно для нестандартных встраиваемых систем.

1.4.3. Отладчики

Отладчик позволяет разработчикам контролировать выполнение программы и просматривать его, чтобы находить и исправлять ошибки. Он предоставляет такие возможности, как установка точек останова, пошаговое выполнение кода построчно и просмотр значений переменных и состояния памяти. Отладчики необходимы для тщательной разработки и поддержки программного обеспечения: они помогают убедиться, что код ведёт себя так, как задумано, в различных условиях.

Отладчики умеют:
- работать с регистрами, отображёнными в память;
- устанавливать точки останова и точки наблюдения (breakpoints/watchpoints);
- читать и записывать регистры, отображённые в память;
- определять, что MCU остановлен из-за отладочного события;
- продолжать выполнение MCU после отладочного события;
- стирать и записывать флеш-память микроконтроллера.

1.4.3.1. Расширение Probe-rs для Visual Studio Code

У Probe-rs есть расширение для Visual Studio Code, которое обеспечивает удобную отладку без обширной настройки. Через это подключение разработчики могут использовать возможности, специфичные для Rust, например красивый вывод значений (pretty printing) и подробные сообщения об ошибках, так что процесс отладки соответствует экосистеме Rust.

1.4.3.2. TRACE32

TRACE32 — профессиональное решение для отладки и трассировки, разработанное компанией Lauterbach для встраиваемых систем. Оно поддерживает широкий спектр архитектур процессоров, включая ARM и RISC-V, и подключается к целевому оборудованию через JTAG, SWD и различные интерфейсы трассировки. TRACE32 предоставляет продвинутые возможности отладки, такие как отладка многоядерных систем, сложные точки останова и анализ трассировки в реальном времени. Он работает со стандартной отладочной информацией ELF/DWARF, поэтому совместим с бинарными файлами Rust, собранными обычными тулчейнами.

1.4.3.3. GDB (GNU Debugger)

GDB — универсальный отладчик, который позволяет разработчикам исследовать состояние программ во время их работы или после падения. Для встраиваемого Rust GDB подключается к целевой системе через OpenOCD или другие серверы отладки, чтобы взаимодействовать со встраиваемым кодом. GDB настраивается очень гибко и поддерживает такие возможности, как удалённая отладка, просмотр переменных и условные точки останова. Его можно использовать на разных платформах, и он имеет обширную поддержку специфичных для Rust задач отладки, например красивого вывода значений и интеграции с IDE.

1.4.4. Зонды

Аппаратный зонд — устройство, которое используется при разработке и отладке встраиваемых систем для обмена данными между компьютером-хостом и встраиваемым устройством. Обычно он поддерживает такие протоколы, как JTAG или SWD, что позволяет ему программировать, отлаживать и анализировать микроконтроллер или микропроцессор встраиваемой системы. Аппаратные зонды помогают разработчикам ставить точки останова, пошагово выполнять код и исследовать память и регистры процессора, что позволяет диагностировать и исправлять проблемы в реальном времени.

1.4.4.1. Rusty-probe

Rusty-probe — USB-зонд для аппаратной отладки с открытым исходным кодом, который предназначен для работы с probe-rs. Связка Rusty-probe и probe-rs представляет собой простое и недорогое решение для разработчиков встраиваемых приложений на Rust.

1.4.4.2. ST-Link

ST-Link — популярный зонд для отладки и программирования, разработанный STMicroelectronics в первую очередь для микроконтроллеров серий STM32 и STM8. Он поддерживает как отладку, так и программирование через интерфейсы JTAG или SWD (Serial Wire Debug). ST-Link широко используется благодаря прямой поддержке со стороны обширной линейки отладочных плат STMicroelectronics и интеграции в основные IDE, что делает его удобным выбором для разработчиков, работающих с микроконтроллерами STM.

1.4.4.3. J-Link

J-Link, разработанный компанией SEGGER Microcontroller, — надёжный и универсальный отладчик, который поддерживает широкий спектр процессорных ядер и устройств, не только ARM, но и RISC-V. Известный высокой производительностью и надёжностью, J-Link поддерживает различные интерфейсы связи, включая JTAG, SWD и JTAG с малым шагом выводов (fine-pitch JTAG). Его ценят за продвинутые возможности, например неограниченное количество точек останова во флеш-памяти, и за совместимость с большим количеством сред разработки.

1.4.4.4. MCU-Link

MCU-Link — отладочный зонд, который также работает как программатор; его выпускает NXP Semiconductors. Он поддерживает множество микроконтроллеров ARM Cortex и легко интегрируется с инструментами разработки, например с IDE MCUXpresso. MCU-Link особенно примечателен своей универсальностью и доступной ценой, что делает его удобным вариантом для любителей, преподавателей и профессиональных разработчиков.

1.5. Установка инструментов

На этой странице приведены не зависящие от ОС инструкции по установке некоторых инструментов:

1.5.0.1. Тулчейн Rust

Установите rustup, следуя инструкциям на https://rustup.rs.

ПРИМЕЧАНИЕ Убедитесь, что версия компилятора не ниже `1.31`. Команда `rustc -V` должна выводить дату новее указанной ниже.

```
$ rustc -V
rustc 1.31.1 (b6c32da9b 2018-12-18)
```

Для экономии трафика и места на диске стандартная установка поддерживает только нативную компиляцию. Чтобы добавить поддержку кросс-компиляции для архитектур ARM Cortex-M, выберите одну из следующих целевых платформ (compilation targets). Для платы STM32F3DISCOVERY, которую мы используем в примерах этой книги, нужна платформа `thumbv7em-none-eabihf`. Выберите подходящий для вас Cortex-M.

Cortex-M0, M0+ и M1 (архитектура ARMv6-M):

```
rustup target add thumbv6m-none-eabi
```

Cortex-M3 (архитектура ARMv7-M):

```
rustup target add thumbv7m-none-eabi
```
Shell

Cortex-M4 и M7 без аппаратной поддержки операций с плавающей точкой (архитектура ARMv7E-M):

```
rustup target add thumbv7em-none-eabi
```
Shell

Cortex-M4F и M7F с аппаратной поддержкой операций с плавающей точкой (архитектура ARMv7E-M):

```
rustup target add thumbv7em-none-eabihf
```
Shell

Cortex-M23 (архитектура ARMv8-M):

```
rustup target add thumbv8m.base-none-eabi
```
Shell

Cortex-M33 и M35P (архитектура ARMv8-M):

```
rustup target add thumbv8m.main-none-eabi
```
Shell

Cortex-M33F и M35PF с аппаратной поддержкой операций с плавающей точкой (архитектура ARMv8-M):

```
rustup target add thumbv8m.main-none-eabihf
```
Shell

1.5.0.2. `cargo-binutils`

```
cargo install cargo-binutils
```
Shell

```
rustup component add llvm-tools
```

WINDOWS: предполагается, что установлены C++ Build Tools для Visual Studio 2019. https://visualstudio.microsoft.com/thank-you-downloading-visual-studio/?sku=BuildTools&rel=16

1.5.0.3. `cargo-generate`

Мы воспользуемся этим инструментом позже, чтобы создать проект из шаблона.

```
cargo install cargo-generate
```
Shell

Примечание: в некоторых дистрибутивах Linux (например, Ubuntu) перед установкой cargo-generate может понадобиться установить пакеты `libssl-dev` и `pkg-config`.

1.5.0.4. Инструкции для конкретной ОС

Теперь следуйте инструкциям для вашей операционной системы:

- Linux
- Windows
- macOS

1.5.1. Linux

Ниже приведены команды установки для нескольких дистрибутивов Linux.

1.5.1.1. Пакеты

- Ubuntu 18.04 или новее / Debian stretch или новее

ПРИМЕЧАНИЕ `gdb-multiarch` — команда GDB, которую вы будете использовать для отладки программ ARM Cortex-M

```
sudo apt install gdb-multiarch openocd qemu-system-arm
```
Shell

- Ubuntu 14.04 и 16.04

ПРИМЕЧАНИЕ `arm-none-eabi-gdb` — команда GDB, которую вы будете использовать для отладки программ ARM Cortex-M

```
sudo apt install gdb-arm-none-eabi openocd qemu-system-arm
```
Shell

- Fedora 27 или новее

```
sudo dnf install gdb openocd qemu-system-arm
```
Shell

- Arch Linux

ПРИМЕЧАНИЕ `arm-none-eabi-gdb` — команда GDB, которую вы будете использовать для отладки программ ARM Cortex-M

```
sudo pacman -S arm-none-eabi-gdb qemu-system-arm openocd
```
Shell

1.5.1.2. Правила udev

Это правило позволяет использовать OpenOCD с отладочной платой без прав root.

Создайте файл `/etc/udev/rules.d/70-st-link.rules` с содержимым, показанным ниже.

```
# STM32F3DISCOVERY rev A/B - ST-LINK/V2
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", TAG+="uaccess"

# STM32F3DISCOVERY rev C+ - ST-LINK/V2-1
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", TAG+="uaccess"
```
Plain Text

Затем перезагрузите все правила udev командой:

```
sudo udevadm control --reload-rules
```
Shell

Если плата была подключена к ноутбуку, отключите её и подключите снова.

Проверить права доступа можно командой:

```
lsusb
```
Shell

Результат должен выглядеть примерно так:

```
(..)
Bus 001 Device 018: ID 0483:374b STMicroelectronics ST-LINK/V2.1
(..)
```
Plain Text

Запомните номера шины и устройства. Используйте их, чтобы составить путь вида `/dev/bus/usb/<bus>/<device>`. Затем используйте этот путь так:

```
ls -l /dev/bus/usb/001/018
```
Shell

```
crw-------+ 1 root root 189, 17 Sep 13 12:34 /dev/bus/usb/001/018
```
Plain Text

```
getfacl /dev/bus/usb/001/018 | grep user
```
Shell

```
user::rw-
user:you:rw-
```
Plain Text

Знак `+` в конце прав доступа означает наличие расширенных прав (extended permission). Команда `getfacl` показывает, что пользователь `you` (ваш пользователь) может использовать это устройство.

Теперь перейдите к следующему разделу.

1.5.2. macOS

Все инструменты можно установить с помощью Homebrew или MacPorts:

1.5.2.1. Установка инструментов через Homebrew

```
$ # GDB
$ brew install arm-none-eabi-gdb

$ # OpenOCD
$ brew install openocd

$ # QEMU
$ brew install qemu
```
Plain Text

> **ПРИМЕЧАНИЕ** Если OpenOCD аварийно завершается, может понадобиться установить последнюю версию командой:

```
$ brew install --HEAD openocd
```
Plain Text

1.5.2.2. Установка инструментов через MacPorts

```
$ # GDB
$ sudo port install arm-none-eabi-gcc

$ # OpenOCD
$ sudo port install openocd

$ # QEMU
$ sudo port install qemu
```
Plain Text

Всё! Перейдите к следующему разделу.

1.5.3. Windows

1.5.3.1. `arm-none-eabi-gdb`

ARM предоставляет установщики `.exe` для Windows. Скачайте один из них здесь и следуйте инструкциям. Перед самым завершением установки отметьте опцию «Add path to environ-

ment variable» (добавить путь в переменную окружения). Затем убедитесь, что инструменты доступны в вашем `%PATH%`:

```
$ arm-none-eabi-gdb -v
GNU gdb (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 8.1.0.20180315-git
(..)
```

1.5.3.2. OpenOCD

Официального бинарного релиза OpenOCD для Windows нет. Но если вам не хочется собирать его самостоятельно, проект xPack предоставляет готовую бинарную сборку, здесь. Следуйте приведённым там инструкциям по установке. Затем обновите переменную окружения `%PATH%`, добавив в неё путь, по которому установлены исполняемые файлы (`C:\Users\USERNAME\AppData\Roaming\xPacks\@xpack-dev-tools\openocd\0.10.0-13.1\.content\bin\`, если вы использовали простую установку).

Убедитесь, что OpenOCD доступен в вашем `%PATH%`:

```
$ openocd -v
Open On-Chip Debugger 0.10.0
(..)
```

1.5.3.3. QEMU

Скачайте QEMU с официального сайта.

1.5.3.4. Драйвер USB для ST-LINK

Также вам нужно установить этот USB-драйвер, иначе OpenOCD не будет работать. Следуйте инструкциям установщика и убедитесь, что вы выбрали подходящую версию драйвера (32-битную или 64-битную).

Всё! Перейдите к следующему разделу.

1.5.4. Проверка установки

В этом разделе мы проверим, что нужные инструменты и драйверы установлены и настроены правильно.

Подключите ноутбук или ПК к отладочной плате с помощью кабеля Mini-USB. На отладочной плате есть два USB-разъёма; используйте тот, что подписан «USB ST-LINK» и находится в центре края платы.

Также проверьте, что разъём ST-LINK установлен. См. изображение ниже: разъём ST-LINK выделен.

Теперь выполните следующую команду:

```
openocd -f interface/stlink.cfg -f target/stm32f3x.cfg
```

> **ПРИМЕЧАНИЕ**: Старые версии openocd, включая релиз 0.10.0 от 2017 года, не содержат нового (и предпочтительного) файла `interface/stlink.cfg`; вместо него может понадобиться использовать `interface/stlink-v2.cfg` или `interface/stlink-v2-1.cfg`.

Вы должны получить примерно следующий вывод, и программа должна продолжать работать в консоли, не возвращая приглашение:

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use 'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.919881
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Содержимое может не совпадать в точности, но последняя строка о breakpoints и watchpoints должна присутствовать. Если она есть, завершите процесс OpenOCD и перейдите к следующему разделу.

Если строки про «breakpoints» нет, попробуйте одну из следующих команд.

```
openocd -f interface/stlink-v2.cfg -f target/stm32f3x.cfg
```

```
openocd -f interface/stlink-v2-1.cfg -f target/stm32f3x.cfg
```

Если одна из этих команд работает, значит, у вас старая ревизия отладочной платы. Это не проблема, но запомните этот факт: позже вам понадобится немного иначе настроить некоторые вещи. Можете перейти к следующему разделу.

Если ни одна из команд не работает от имени обычного пользователя, попробуйте запустить их с правами root (например, `sudo openocd ..`). Если с правами root команды работают, проверьте, что правила udev настроены правильно.

Если вы дошли до этого места и OpenOCD всё ещё не работает, пожалуйста, откройте задачу (issue) — мы вам поможем!

2. Начало работы

В этом разделе мы покажем, как писать, собирать, прошивать и отлаживать встраиваемые программы. Большую часть примеров можно попробовать без специального оборудования: основы мы покажем с помощью QEMU, популярного эмулятора аппаратного обеспечения с открытым исходным кодом. Единственный раздел, где оборудование действительно нужно, — это Оборудование, где мы используем OpenOCD, чтобы прошить STM32F3DISCOVERY.

2.1. QEMU

Мы начнём писать программу для LM3S6965 — микроконтроллера Cortex-M3. Мы выбрали его в качестве первой цели, потому что его можно эмулировать с помощью QEMU. Поэтому в этом разделе вам не придётся возиться с железом, и мы сможем сосредоточиться на инструментах и процессе разработки.

ВАЖНО В этом руководстве мы будем называть проект «app». Всякий раз, когда вы видите слово «app», замените его на имя, которое вы выбрали для своего проекта. Либо назовите проект «app», и подстановки не понадобятся.

2.1.1. Создание нестандартной программы на Rust

Мы используем шаблон проекта `cortex-m-quickstart`, чтобы создать из него новый проект. Созданный проект будет содержать минимальное приложение — хорошую отправную точку для нового встраиваемого приложения на Rust. Кроме того, проект будет содержать каталог `examples` с несколькими отдельными приложениями, которые демонстрируют ключевые возможности встраиваемого Rust.

2.1.1.1. Использование `cargo-generate`

Сначала установите cargo-generate:

```
cargo install cargo-generate
```
Shell

Затем создайте новый проект:

```
cargo generate --git https://github.com/knurling-rs/app-template
```
Shell

```
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app
```
Plain Text

```
cd app
```
Shell

2.1.1.2. Использование `git`

Клонируйте репозиторий:

```
git clone https://github.com/rust-embedded/cortex-m-quickstart app
cd app
```
Shell

Затем заполните плейсхолдеры в файле `Cargo.toml`:

```
[package]
authors = ["{{authors}}"] # "{{authors}}" -> "John Smith"
edition = "2018"
name = "{{project-name}}" # "{{project-name}}" -> "app"
version = "0.1.0"
```
TOML

```
# ..

[[bin]]
name = "{{project-name}}" # "{{project-name}}" -> "app"
test = false
bench = false
```

2.1.1.3. Без использования инструментов

Скачайте последний снимок шаблона `cortex-m-quickstart` и распакуйте его.

```
curl -LO https://github.com/rust-embedded/cortex-m-quickstart/archive/
master.zip
unzip master.zip
mv cortex-m-quickstart-master app
cd app
```

Shell

Или вы можете открыть страницу `cortex-m-quickstart`, нажать зелёную кнопку «Clone or download», а затем «Download ZIP».

Затем заполните плейсхолдеры в файле `Cargo.toml`, как во второй части варианта «Использование `git`».

2.1.2. Обзор программы

Для удобства здесь приведены самые важные части исходного кода из `src/main.rs`:

```
#![no_std]
#![no_main]

use panic_halt as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {
        // здесь будет ваш код
    }
}
```

Rust

Эта программа немного отличается от стандартной программы на Rust, поэтому давайте присмотримся к ней внимательнее.

`#![no_std]` указывает, что программа *не* будет связываться со стандартным крейтом `std`. Вместо этого она будет связываться с его подмножеством — крейтом `core`.

`#![no_main]` указывает, что программа не будет использовать стандартный интерфейс `main`, который применяют большинство программ на Rust. Главная причина использовать `no_main` (тут без каламбура) в том, что интерфейс `main` в контексте `no_std` требует nightly-версии компилятора.

`use panic_halt as _;`. Этот крейт предоставляет `panic_handler`, который определяет поведение программы при панике. Мы подробнее рассмотрим это в главе Паника этой книги.

`#[entry]` — атрибут, который предоставляет крейт `cortex-m-rt` и который используется, чтобы пометить точку входа программы. Поскольку мы не используем стандартный интерфейс `main`, нам нужен другой способ указать точку входа, и таким способом является `#[entry]`.

`fn main() -> !`. Наша программа будет *единственным* процессом, работающим на целевом оборудовании, поэтому мы не хотим, чтобы она завершалась! Мы используем расходящуюся функцию (часть `-> !` в сигнатуре функции), чтобы на этапе компиляции гарантировать это.

2.1.3. Кросс-компиляция

Прежде всего нам понадобится раскладка памяти для целевого микроконтроллера — в нашем случае LM3S6965. Без неё сборка не сможет скомпоновать образ. Создайте в корне проекта файл `memory.x` и вставьте в него следующее содержимое:

```
MEMORY                                                          Plain Text
{
  /* ЗАМЕЧАНИЕ: 1 K = 1 KiBi = 1024 байта */
  /* TODO: подстройте эти области памяти под раскладку памяти вашего устройства */
  /* Эти значения соответствуют LM3S6965 — одному из немногих устройств, которые
  может эмулировать QEMU */
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}

/* Здесь будет выделен стек вызовов. */
/* Стек имеет полный нисходящий (full descending) тип. */
/* Эту переменную можно использовать, чтобы разместить стек вызовов и статические
  переменные в разных областях памяти. Ниже показано значение по умолчанию */
/* _stack_start = ORIGIN(RAM) + LENGTH(RAM); */

/* Этим символом можно задать расположение секции .text */
/* Если его не указывать, секция .text будет размещена сразу после
секции .vector_table */
/* Это требуется только на микроконтроллерах, которые хранят некоторую конфигурацию
  сразу после таблицы векторов */
/* _stext = ORIGIN(FLASH) + 0x400; */

/* Пример размещения неинициализированных переменных в произвольных областях RAM. */
/* Предполагается, что вы определили область RAM2 выше, а в исходниках на Rust
  добавили атрибут `#[link_section = ".ram2bss"]` к данным,
  которые хотите там разместить. */
/* Обратите внимание, что эта секция не будет обнулена рантаймом! */
/* SECTIONS {
     .ram2bss (NOLOAD) : ALIGN(4) {
       *(.ram2bss);
       . = ALIGN(4);
     } > RAM2
```

```
   } INSERT AFTER .bss;
*/
```

Следующий шаг — *кросс*-компиляция программы для архитектуры Cortex-M3. Это так же просто, как выполнить `cargo build --target $TRIPLE`, если вы знаете, какой должна быть целевая платформа (`$TRIPLE`). К счастью, `.cargo/config.toml` в шаблоне уже содержит ответ:

```
tail -n6 .cargo/config.toml
```
Shell

```
[build]
# Выберите ОДНУ из этих целевых платформ
# target = "thumbv6m-none-eabi"    # Cortex-M0 и Cortex-M0+
target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4 и Cortex-M7 (no FPU)
# target = "thumbv7em-none-eabihf" # Cortex-M4F и Cortex-M7F (with FPU)
```
TOML

Для кросс-компиляции под архитектуру Cortex-M3 нужно использовать `thumbv7m-none-eabi`. Эта целевая платформа не устанавливается автоматически вместе с тулчейном Rust, поэтому сейчас самое время добавить его в тулчейн, если вы ещё этого не сделали:

```
rustup target add thumbv7m-none-eabi
```
Shell

Поскольку целевая платформа `thumbv7m-none-eabi` указана по умолчанию в вашем `.cargo/config.toml`, две команды ниже делают одно и то же:

```
cargo build --target thumbv7m-none-eabi
cargo build
```
Shell

2.1.4. Инспекция

Теперь у нас есть ELF-бинарный файл, не нативный для вашей машины, в `target/thumbv7m-none-eabi/debug/app`. Мы можем исследовать его с помощью `cargo-binutils`.

С помощью `cargo-readobj` мы можем вывести заголовки ELF, чтобы убедиться, что это бинарный файл для ARM.

```
cargo readobj --bin app -- --file-headers
```
Shell

Обратите внимание:

- `--bin app` — сокращение для инспекции бинарного файла `target/$TRIPLE/debug/app`;
- `--bin app` также (пере)компилирует бинарный файл, если это необходимо

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0x0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
```
Plain Text

```
  Entry point address:               0x405
  Start of program headers:          52 (bytes into file)
  Start of section headers:          153204 (bytes into file)
  Flags:                             0x5000200
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         19
  Section header string table index: 18
```

`cargo-size` может вывести размеры секций компоновщика бинарного файла.

```
cargo size --bin app --release -- -A
```
Shell

здесь мы используем `--release`, чтобы посмотреть оптимизированную версию

```
app  :
section              size        addr
.vector_table        1024         0x0
.text                  92       0x400
.rodata                 0       0x45c
.data                   0  0x20000000
.bss                    0  0x20000000
.debug_str           2958         0x0
.debug_loc             19         0x0
.debug_abbrev         567         0x0
.debug_info          4929         0x0
.debug_ranges          40         0x0
.debug_macinfo          1         0x0
.debug_pubnames      2035         0x0
.debug_pubtypes      1892         0x0
.ARM.attributes        46         0x0
.debug_frame          100         0x0
.debug_line           867         0x0
Total               14570
```
Plain Text

Напоминание о секциях компоновщика ELF

- `.text` содержит инструкции программы
- `.rodata` содержит константы, например строки
- `.data` содержит статически выделенные переменные, начальные значения которых *не* равны нулю
- `.bss` также содержит статически выделенные переменные, но их начальные значения равны нулю
- `.vector_table` — *нестандартная* секция, которую мы используем для хранения таблицы векторов (прерываний)
- `.ARM.attributes` и секции `.debug_*` содержат метаданные и *не* загружаются на целевое устройство при прошивке бинарного файла.

ВАЖНО: файлы ELF содержат метаданные, например отладочную информацию, поэтому их *размер на диске не* отражает точно объём памяти, который программа займёт на устройстве после прошивки. *Всегда* используйте `cargo-size`, чтобы проверить, насколько большой бинарный файл на самом деле.

С помощью `cargo-objdump` можно выполнить дизассемблирование бинарного файла.

```
cargo objdump --bin app --release -- --disassemble --no-show-raw-insn --
print-imm-hex
```
Shell

> **ПРИМЕЧАНИЕ** если приведённая выше команда выдаёт `Unknown command line argument`, см. следующий баг-репорт: https://github.com/rust-embedded/book/issues/269

> **ПРИМЕЧАНИЕ** этот вывод может отличаться в вашей системе. Новые версии rustc, LLVM и библиотек могут генерировать другой ассемблерный код. Мы сократили некоторые инструкции, чтобы фрагмент оставался компактным.

```
app:  file format ELF32-arm-little

Disassembly of section .text:
main:
     400: bl  #0x256
     404: b #-0x4 <main+0x4>

Reset:
     406: bl  #0x24e
     40a: movw  r0, #0x0
     < .. truncated any more instructions .. >

DefaultHandler_:
     656: b #-0x4 <DefaultHandler_>

UsageFault:
     657: strb  r7, [r4, #0x3]

DefaultPreInit:
     658: bx  lr

__pre_init:
     659: strb  r7, [r0, #0x1]

__nop:
     65a: bx  lr

HardFaultTrampoline:
     65c: mrs r0, msp
     660: b #-0x2 <HardFault_>
```
Plain Text

```
HardFault_:
     662: b #-0x4 <HardFault_>

HardFault:
     663: <unknown>
```

2.1.5. Запуск

Теперь посмотрим, как запустить встраиваемую программу на QEMU! На этот раз мы используем пример `hello`, который уже что-то делает. По умолчанию этот пример использует `[defmt]` и RTT, чтобы выводить текст.

> **ПРИМЕЧАНИЕ** `defmt` — сторонняя зависимость (то есть не входящая в core), широко используемая в экосистеме Embedded Rust.

Чтобы читать и декодировать сообщения, которые `defmt` выдаёт на хосте, нам нужно переключить транспорт вывода RTT на semihosting. При работе с реальным оборудованием для этого нужна отладочная сессия, а при использовании QEMU всё работает само собой.

Переключим зависимости:

```shell
cargo remove defmt-rtt
cargo add defmt-semihosting
```

Откройте `src/lib.rs` и замените `use defmt_rtt as _;` на `use defmt_semihosting as _;`

Теперь можно собрать пример:

```shell
cargo build --bin hello
```

Выходной бинарный файл появится по пути `target/thumbv7m-none-eabi/debug/hello`.

Чтобы запустить этот бинарный файл на QEMU, обычно достаточно такой команды:

```shell
qemu-system-arm \
  -cpu cortex-m3 \
  -machine lm3s6965evb \
  -nographic \
  -semihosting-config enable=on,target=native \
  -kernel target/thumbv7m-none-eabi/debug/hello
```

В нашем случае, поскольку мы используем `defmt`, хост не сможет декодировать вывод. Вместо этого нам понадобится инструмент от Ferrous Systems под названием `qemu-run`:

```shell
git clone git@github.com:knurling-rs/defmt.git
cd defmt/qemu-run/
cargo run -- --machine lm3s6965evb ../qemu-rs/target/thumbv7m-none-eabi/debug/hello
```

```
Hello, world!
```

Команда должна успешно завершиться (код выхода = 0) после вывода текста. В *nix-системах это можно проверить такой командой:

```
echo $?
```
Shell

```
0
```
Plain Text

Разберём эту команду QEMU по частям:

- `qemu-system-arm`. Это эмулятор QEMU. Существует несколько вариантов этих бинарных файлов QEMU; этот выполняет полную *системную* эмуляцию машин *ARM*, отсюда и название.
- `-cpu cortex-m3`. Указывает QEMU эмулировать процессор Cortex-M3. Указание модели процессора позволяет поймать некоторые ошибки компиляции: например, запуск программы, скомпилированной для Cortex-M4F, у которого есть аппаратный FPU, приведёт к ошибке QEMU во время выполнения.
- `-machine lm3s6965evb`. Указывает QEMU эмулировать LM3S6965EVB — отладочную плату, на которой стоит микроконтроллер LM3S6965.
- `-nographic`. Указывает QEMU не запускать графический интерфейс.
- `-semihosting-config (..)`. Указывает QEMU включить semihosting. Semihosting позволяет эмулируемому устройству, среди прочего, использовать stdout, stderr и stdin хоста, а также создавать файлы на хосте.
- `-kernel $file`. Указывает QEMU, какой бинарный файл загрузить и запустить на эмулируемой машине.

Вводить такую длинную команду QEMU слишком утомительно! Мы можем задать пользовательский runner, чтобы упростить процесс. В `.cargo/config.toml` есть закомментированный runner, который запускает QEMU; раскомментируем его:

```
head -n3 .cargo/config.toml
```
Shell

```
[target.thumbv7m-none-eabi]
# раскомментируйте, чтобы `cargo run` запускал программы на QEMU
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"
```
TOML

Этот runner применяется только к целевой платформе `thumbv7m-none-eabi`, которая у нас выбрана по умолчанию. Теперь `cargo run` будет компилировать программу и запускать её на QEMU:

```
cargo ru(
  level: 2 + whole
)--release
```
Shell

```
   Compiling app v0.1.0 (file:///tmp/app)
    Finished release [optimized + debuginfo] target(s) in 0.26s
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
     semihosting-config enable=on,target=native -kernel target/thumbv7m-none-eabi/
     release/examples/hello`
Hello, world!
```
Plain Text

2.1.6. Отладка

Отладка критически важна для разработки встраиваемых систем. Давайте посмотрим, как её выполнять.

Отладка встраиваемого устройства подразумевает *удалённую* отладку, потому что программа, которую мы хотим отлаживать, не будет работать на той же машине, где запущен отладчик (GDB или LLDB).

Удалённая отладка включает клиента и сервер. В конфигурации с QEMU клиентом будет процесс GDB (или LLDB), а сервером — процесс QEMU, который также запускает встраиваемую программу.

В этом разделе мы будем использовать уже собранный пример `hello`.

Первый шаг отладки — запустить QEMU в режиме отладки:

```
qemu-system-arm \                                                    Shell
  -cpu cortex-m3 \
  -machine lm3s6965evb \
  -nographic \
  -semihosting-config enable=on,target=native \
  -gdb tcp::3333 \
  -S \
  -kernel target/thumbv7m-none-eabi/debug/examples/hello
```

Эта команда ничего не выведет в консоль и заблокирует терминал. На этот раз мы передали два дополнительных флага:

- `-gdb tcp::3333`. Указывает QEMU ждать подключения GDB по TCP-порту 3333.
- `-S`. Указывает QEMU заморозить машину при запуске. Без этого программа дошла бы до конца main раньше, чем у нас появилась бы возможность подключить отладчик!

Далее запустите GDB в другом терминале и укажите ему загрузить отладочные символы примера:

```
gdb-multiarch -q target/thumbv7m-none-eabi/debug/examples/hello      Shell
```

ПРИМЕЧАНИЕ: в зависимости от того, что вы установили в главе об установке, вместо `gdb-multiarch` может понадобиться другая версия gdb. Это также может быть `arm-none-eabi-gdb` или просто `gdb`.

Затем в оболочке GDB подключаемся к QEMU, который ждёт подключения на TCP-порту 3333.

```
target remote :3333                                                  Shell
```

```
Remote debugging using :3333                                         Plain Text
Reset () at $REGISTRY/cortex-m-rt-0.6.1/src/lib.rs:473
473     pub unsafe extern "C" fn Reset() -> ! {
```

Вы увидите, что процесс остановлен, а счётчик команд указывает на функцию с именем `Reset`. Это обработчик сброса — то, что ядра Cortex-M выполняют при загрузке.

> Обратите внимание, что в некоторых конфигурациях вместо строки `Reset () at $REGISTRY/cortex-m-rt-0.6.1/src/lib.rs:473`, как показано выше, gdb может выводить предупреждения вида:
>
> `core::num::bignum::Big32x40::mul_small () at src/libcore/num/bignum.rs:254 src/libcore/num/bignum.rs: No such file or directory.`

> Это известная особенность. Такие предупреждения можно спокойно игнорировать: скорее всего, вы находитесь в `Reset()`.

Этот обработчик сброса в итоге вызовет нашу функцию main. Давайте перейдём туда с помощью точки останова и команды `continue`. Чтобы установить точку останова, сначала посмотрим, где бы мы хотели остановиться в коде, с помощью команды `list`.

```
list main
```
Shell

Это покажет исходный код из файла examples/hello.rs.

```
6       use panic_halt as _;
7
8       use cortex_m_rt::entry;
9       use cortex_m_semihosting::{debug, hprintln};
10
11      #[entry]
12      fn main() -> ! {
13          hprintln!("Hello, world!").unwrap();
14
15          // exit QEMU
```
Plain Text

Мы хотим добавить точку останова непосредственно перед «Hello, world!», то есть на строке 13. Делаем это командой `break`:

```
break 13
```
Shell

Теперь можно велеть gdb выполнить код до нашей функции main командой `continue`:

```
continue
```
Shell

```
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main () at examples\hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```
Plain Text

Мы находимся близко к коду, который выводит «Hello, world!». Продвинемся дальше командой `next`.

```
next
```
Shell

```
16          debug::exit(debug::EXIT_SUCCESS);
```
Plain Text

В этот момент вы должны увидеть «Hello, world!», выведенное в терминале, где запущен `qemu-system-arm`.

```
$ qemu-system-arm (..)
Hello, world!
```
Plain Text

Повторный вызов `next` завершит процесс QEMU.

```
next
```
Shell

```
[Inferior 1 (Remote target) exited normally]
```
Plain Text

Теперь можно выйти из сеанса GDB.

```
quit
```
Shell

3. Оборудование

К этому моменту вы должны уже немного освоить инструменты и процесс разработки. В этом разделе мы перейдём на настоящее оборудование; процесс в основном останется прежним. Давайте начнём.

3.0.1. Знакомство с вашим оборудованием

Прежде чем начать, нужно выяснить некоторые характеристики целевого устройства, так как они понадобятся для настройки проекта:

- Ядро ARM, например Cortex-M3.
- Есть ли в ядре ARM FPU? Ядра Cortex-M4**F** и Cortex-M7**F** его содержат.
- Сколько флеш-памяти и ОЗУ есть в целевом устройстве? Например, 256 КиБ флеш-памяти и 32 КиБ ОЗУ.
- Где в адресном пространстве отображены флеш-память и ОЗУ? Например, ОЗУ обычно расположено по адресу `0x2000_0000`.

Эту информацию можно найти в техническом описании (data sheet) или справочном руководстве (reference manual) вашего устройства.

В этом разделе мы будем использовать наше эталонное оборудование — STM32F3DISCOVERY. Эта плата содержит микроконтроллер STM32F303VCT6. Он имеет:

- ядро Cortex-M4F, которое включает FPU одинарной точности;
- 256 КиБ флеш-памяти по адресу `0x0800_0000`;
- 40 КиБ ОЗУ по адресу `0x2000_0000`. (Есть ещё одна область ОЗУ, но для простоты мы её учитывать не будем.)

3.0.2. Настройка

Начнём с нуля, с нового экземпляра шаблона. См. предыдущий раздел про QEMU — там есть напоминание о том, как сделать это без `cargo-generate`.

```
$ cargo generate --git https://github.com/rust-embedded/cortex-m-quickstart
 Project Name: app
 Creating project called `app`...
 Done! New project created /tmp/app

$ cd app
```

Первый шаг — задать целевую платформу по умолчанию в `.cargo/config.toml`.

```
tail -n5 .cargo/config.toml
```

```
# Выберите ОДНУ из этих целевых платформ
# target = "thumbv6m-none-eabi"    # Cortex-M0 и Cortex-M0+
# target = "thumbv7m-none-eabi"    # Cortex-M3
# target = "thumbv7em-none-eabi"   # Cortex-M4 и Cortex-M7 (без FPU)
target = "thumbv7em-none-eabihf" # Cortex-M4F и Cortex-M7F (с FPU)
```

Мы будем использовать `thumbv7em-none-eabihf`, так как эта платформа подходит для ядра Cortex-M4F.

> **ПРИМЕЧАНИЕ**: как вы помните из предыдущей главы, каждую целевую платформу нужно устанавливать, а эта — новая. Поэтому не забудьте выполнить `rustup target add thumbv7em-none-eabihf`.

Второй шаг — внести информацию об областях памяти в файл `memory.x`.

```
$ cat memory.x
/* Скрипт компоновщика для STM32F303VCT6 */
MEMORY
{
  /* ЗАМЕЧАНИЕ: 1 K = 1 KiBi = 1024 байта */
  FLASH : ORIGIN = 0x08000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 40K
}
```

> **ПРИМЕЧАНИЕ**: если по какой-то причине вы изменили файл `memory.x` после первой сборки для конкретной целевой платформы, выполните `cargo clean` перед `cargo build`, так как `cargo build` может не отслеживать изменения в `memory.x`.

Начнём снова с примера hello, но сначала нужно сделать небольшое изменение.

В `examples/hello.rs` убедитесь, что вызов `debug::exit()` закомментирован или удалён. Он нужен только для запуска в QEMU.

```rust
#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    // выход из QEMU
    // ПРИМЕЧАНИЕ: не запускайте это на реальном оборудовании; это может повредить состояние OpenOCD
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Теперь можно кросс-компилировать программы командой `cargo build`, а бинарные файлы исследовать с помощью `cargo-binutils`, как вы делали раньше. Крейт `cortex-m-rt` берёт на себя всю магию, необходимую, чтобы ваш чип заработал: практически все процессоры Cortex-M загружаются одинаковым образом.

```
cargo build --example hello
```

3.0.3. Отладка

Отладка будет выглядеть немного иначе. Более того, первые шаги могут отличаться в зависимости от целевого устройства. В этом разделе мы покажем шаги, необходимые для отладки программы, запущенной на STM32F3DISCOVERY. Это справочный материал; за информацией об отладке конкретного устройства обращайтесь к Debugonomicon.

Как и раньше, мы будем использовать удалённую отладку, а клиентом будет процесс GDB. На этот раз сервером будет OpenOCD.

Как и в разделе проверка установки, подключите отладочную плату к ноутбуку или ПК и проверьте, что разъём ST-LINK установлен.

В терминале запустите `openocd`, чтобы подключиться к ST-LINK на отладочной плате. Запускайте эту команду из корня шаблона: `openocd` найдёт файл `openocd.cfg`, в котором указано, какой файл интерфейса и какой файл целевого устройства использовать.

```
cat openocd.cfg
```
Shell

```
# Пример конфигурации OpenOCD для отладочной платы STM32F3DISCOVERY

# В зависимости от ревизии вашей платы выберите ОДИН из этих интерфейсов.
# В любой момент времени должна быть раскомментирована только одна строка source.

# Ревизия C (более новая)
source [find interface/stlink.cfg]

# Ревизии A и B (более старые)
# source [find interface/stlink-v2.cfg]

source [find target/stm32f3x.cfg]
```
Plain Text

ПРИМЕЧАНИЕ Если на шаге проверки установки вы выяснили, что у вас более старая ревизия отладочной платы, то на этом этапе нужно изменить файл `openocd.cfg`, чтобы использовать `interface/stlink-v2.cfg`.

```
$ openocd
```
Plain Text

```
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
Info : auto-selecting first available session transport "hla_swd". To override use
'transport select <transport>'.
adapter speed: 1000 kHz
adapter_nsrst_delay: 100
Info : The selected transport took over low-level target control. The results might
differ compared to plain JTAG/SWD
none separate
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : Unable to match requested speed 1000 kHz, using 950 kHz
Info : clock speed 950 kHz
Info : STLINK v2 JTAG v27 API v2 SWIM v15 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.913879
Info : stm32f3x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

В другом терминале запустите GDB, тоже из корня шаблона.

```
gdb-multiarch -q target/thumbv7em-none-eabihf/debug/examples/hello
```
Plain Text

ПРИМЕЧАНИЕ: как и раньше, вместо `gdb-multiarch` может понадобиться другая версия gdb — в зависимости от той, что вы установили в главе об установке. Это также может быть `arm-none-eabi-gdb` или просто `gdb`.

Далее подключите GDB к OpenOCD, который ждёт TCP-соединения на порту 3333.

```
(gdb) target remote :3333
Remote debugging using :3333
0x00000000 in ?? ()
```
Shell

Теперь прошейте (загрузите) программу в микроконтроллер командой `load`.

```
(gdb) load
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x1518 lma 0x8000400
Loading section .rodata, size 0x414 lma 0x8001918
Start address 0x08000400, load size 7468
Transfer rate: 13 KB/sec, 2489 bytes/write.
```
Shell

Программа загружена. Она использует semihosting, поэтому перед любым вызовом semihosting нужно сообщить OpenOCD, что semihosting следует включить. Команды для OpenOCD можно отправлять с помощью команды `monitor`.

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```
Shell

> Все команды OpenOCD можно посмотреть, вызвав команду `monitor help`.

Как и раньше, можно перейти сразу к `main` с помощью точки останова и команды `continue`.

```
(gdb) break main
Breakpoint 1 at 0x8000490: file examples/hello.rs, line 11.
Note: automatically using hardware breakpoints for read-only addresses.

(gdb) continue
Continuing.

Breakpoint 1, hello::__cortex_m_rt_main_trampoline () at examples/hello.rs:11
11      #[entry]
```
Shell

> **ПРИМЕЧАНИЕ** Если после команды `continue` GDB блокирует терминал вместо того, чтобы остановиться на точке останова, стоит перепроверить, правильно ли в файле `memory.x` заданы области памяти для вашего устройства (и начало, *и* длина).

Войдите в функцию main командой `step`.

```
(gdb) step
halted: PC: 0x08000496
hello::__cortex_m_rt_main () at examples/hello.rs:13
13          hprintln!("Hello, world!").unwrap();
```

После продвижения программы командой `next` вы должны увидеть «Hello, world!», выведенное в консоли OpenOCD, среди прочего.

```
$ openocd
(..)
Info : halted: PC: 0x08000502
Hello, world!
Info : halted: PC: 0x080004ac
Info : halted: PC: 0x080004ae
Info : halted: PC: 0x080004b0
Info : halted: PC: 0x080004b4
Info : halted: PC: 0x080004b8
Info : halted: PC: 0x080004bc
```

Сообщение выводится только один раз, потому что программа входит в бесконечный цикл, определённый в строке 19: `loop {}`.

Теперь можно выйти из GDB командой `quit`.

```
(gdb) quit
A debugging session is active.

        Inferior 1 [Remote target] will be detached.

Quit anyway? (y or n)
```

Отладка теперь требует ещё нескольких шагов, поэтому мы упаковали их в один скрипт GDB с именем `openocd.gdb`. Файл был создан на шаге `cargo generate` и должен работать без изменений. Давайте взглянем:

```
cat openocd.gdb
```

```
target extended-remote :3333

# выводить демангленные (demangled) символы
set print asm-demangle on

# обнаруживать необработанные исключения, hard fault'ы и паники
break DefaultHandler
break HardFault
break rust_begin_unwind

monitor arm semihosting enable
```

```
load

# запустить процесс, но сразу остановить процессор
stepi
```

Теперь запуск `<gdb> -x openocd.gdb target/thumbv7em-none-eabihf/debug/examples/hello` сразу подключит GDB к OpenOCD, включит semihosting, загрузит программу и запустит процесс.

Кроме того, можно превратить `<gdb> -x openocd.gdb` в пользовательский runner, чтобы `cargo run` собирал программу *и* запускал сеанс GDB. Этот runner уже включён в `.cargo/config.toml`, но закомментирован.

```
head -n10 .cargo/config.toml
```
Shell

```
[target.thumbv7m-none-eabi]
# раскомментируйте, чтобы `cargo run` запускал программы на QEMU
# runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
# раскомментируйте ОДИН из этих трёх вариантов, чтобы `cargo run` запускал сеанс GDB
# выбор зависит от вашей системы
runner = "arm-none-eabi-gdb -x openocd.gdb"
# runner = "gdb-multiarch -x openocd.gdb"
# runner = "gdb -x openocd.gdb"
```
TOML

```
$ cargo run --example hello
(..)
Loading section .vector_table, size 0x400 lma 0x8000000
Loading section .text, size 0x1e70 lma 0x8000400
Loading section .rodata, size 0x61c lma 0x8002270
Start address 0x800144e, load size 10380
Transfer rate: 17 KB/sec, 3460 bytes/write.
(gdb)
```
Plain Text

3.1. Регистры, отображённые в память

Встраиваемые системы не могут обойтись только выполнением обычного кода на Rust и перемещением данных в ОЗУ. Если мы хотим передавать информацию в систему или из неё (будь то мигание светодиодом, обнаружение нажатия кнопки или обмен данными с внешней периферией по какой-либо шине), нам придётся заглянуть в мир периферии и её «регистров, отображённых в память».

Скорее всего, код для доступа к периферии вашего микроконтроллера уже написан и существует на одном из следующих уровней:

- Крейт микроархитектуры — такой крейт содержит полезные подпрограммы, общие для ядра процессора, которое использует ваш микроконтроллер, а также для периферии, одинаковой для всех микроконтроллеров с таким типом ядра. Например, крейт cortex-m предоставляет функции для разрешения и запрета прерываний — они одинаковы для всех

микроконтроллеров на базе Cortex-M. Он также даёт доступ к периферии «SysTick», которая есть во всех микроконтроллерах на базе Cortex-M.
- Peripheral Access Crate (PAC) — такой крейт представляет собой тонкую обёртку над регистрами, определёнными для конкретного номера модели микроконтроллера, который вы используете. Например, tm4c123x для серии Texas Instruments Tiva-C TM4C123 или stm32f30x для серии STM32F30x от ST Microelectronics. Здесь вы будете работать с регистрами напрямую, следуя инструкциям по работе каждой периферии, приведённым в техническом справочном руководстве вашего микроконтроллера.
- HAL-крейт — эти крейты предлагают более удобный API для вашего процессора, часто за счёт реализации общих трейтов, определённых в embedded-hal. Например, такой крейт может предлагать структуру `Serial` с конструктором, который принимает подходящий набор выводов GPIO и скорость передачи (baud rate), и функцию вроде `write_byte` для отправки данных. Подробнее о embedded-hal см. главу Переносимость.
- Крейт платы — эти крейты идут на один шаг дальше HAL-крейтов: они заранее настраивают различную периферию и выводы GPIO под конкретный отладочный комплект или плату, которую вы используете, например stm32f3-discovery для платы STM32F3DISCOVERY.

3.1.1. Крейт платы

Крейт платы — идеальная отправная точка, если вы только начинаете работать со встраиваемым Rust. Он хорошо скрывает аппаратные детали, которые могут быть сложными при изучении этой темы, и делает типовые задачи простыми, например включение и выключение светодиода. Функциональность, которую он предоставляет, сильно различается для разных плат. Поскольку эта книга стремится оставаться независимой от конкретного оборудования, крейты плат в ней рассматриваться не будут.

Если вы хотите поэкспериментировать с платой STM32F3DISCOVERY, настоятельно рекомендуем посмотреть крейт платы stm32f3-discovery, который предоставляет функции для мигания светодиодами платы, доступа к её компасу, Bluetooth и многому другому. Книга Discovery — отличное введение в использование крейта платы.

Но если вы работаете с системой, для которой ещё нет специального крейта платы, или вам нужна функциональность, которой нет в существующих крейтах, читайте дальше: мы начнём «снизу», с крейтов микроархитектуры.

3.1.2. Крейт микроархитектуры

Посмотрим на периферию SysTick, которая есть во всех микроконтроллерах на базе Cortex-M. Низкоуровневый API для неё находится в крейте cortex-m, и использовать его можно так:

```
#![no_std]                                                    Rust
#![no_main]
use cortex_m::peripheral::{syst, Peripherals};
use cortex_m_rt::entry;
use panic_halt as _;

#[entry]
fn main() -> ! {
    let peripherals = Peripherals::take().unwrap();
    let mut systick = peripherals.SYST;
    systick.set_clock_source(syst::SystClkSource::Core);
    systick.set_reload(1_000);
```

```
    systick.clear_current();
    systick.enable_counter();
    while !systick.has_wrapped() {
        // Цикл ожидания
    }

    loop {}
}
```

Функции структуры SYST довольно точно соответствуют функциональности этой периферии, описанной в техническом справочном руководстве ARM. В этом API нет ничего вроде «задержки на X миллисекунд» — такую задержку нам приходится грубо реализовывать самостоятельно, с помощью цикла `while`. Обратите внимание, что обратиться к структуре SYST мы можем только после вызова `Peripherals::take()`. Это специальная подпрограмма, которая гарантирует, что во всей программе существует только одна структура SYST. Подробнее об этом см. раздел Периферия.

3.1.3. Использование Peripheral Access Crate (PAC)

Далеко же мы уйдём в разработке встраиваемого ПО, если будем ограничиваться только базовой периферией, которая есть в каждом Cortex-M. Рано или поздно нам понадобится писать код, специфичный для конкретного микроконтроллера, который мы используем. В этом примере предположим, что у нас есть Texas Instruments TM4C123 — довольно обычный Cortex-M4 с частотой 80 МГц и 256 КиБ флеш-памяти. Чтобы работать с этим чипом, подключим крейт tm4c123x.

```rust
#![no_std]
#![no_main]

use panic_halt as _; // обработчик паники

use cortex_m_rt::entry;
use tm4c123x;

#[entry]
pub fn init() -> (Delay, Leds) {
    let cp = cortex_m::Peripherals::take().unwrap();
    let p = tm4c123x::Peripherals::take().unwrap();

    let pwm = p.PWM0;
    pwm.ctl.write(|w| w.globalsync0().clear_bit());
    // Mode = 1 => режим счёта вверх/вниз
    pwm._2_ctl.write(|w| w.enable().set_bit().mode().set_bit());
    pwm._2_gena.write(|w| w.actcmpau().zero().actcmpad().one());
    // 528 тактов (264 вверх и вниз) = 4 цикла на строку видео (2112 тактов)
    pwm._2_load.write(|w| unsafe { w.load().bits(263) });
    pwm._2_cmpa.write(|w| unsafe { w.compa().bits(64) });
    pwm.enable.write(|w| w.pwm4en().set_bit());
}
```

Мы обратились к периферии `PWM0` точно так же, как раньше обращались к `SYST`, только вызвали `tm4c123x::Peripherals::take()`. Поскольку этот крейт сгенерирован с помощью svd2rust, функции доступа к полям наших регистров принимают замыкание, а не числовой аргумент. Хотя кода получается много, компилятор Rust может использовать его, чтобы выполнить за нас множество проверок, а затем сгенерировать машинный код, который очень близок к написанному вручную ассемблеру! Если сгенерированный код не может определить, что все возможные аргументы функции доступа допустимы (например, если SVD определяет регистр как 32-битный, но не указывает, что у некоторых из этих 32-битных значений есть особый смысл), функция помечается как `unsafe`. Мы видим это в примере выше, когда задаём подполя `load` и `compa` с помощью функции `bits()`.

3.1.3.1. Чтение

Функция `read()` возвращает объект, который даёт доступ только для чтения к различным подполям этого регистра, как они определены в файле SVD производителя для данного чипа. Все функции, доступные для специального типа возврата `R` этого регистра, в этой периферии, на этом чипе, можно найти в документации tm4c123x.

```rust
if pwm.ctl.read().globalsync0().is_set() {
    // Делаем что-то
}
```

3.1.3.2. Запись

Функция `write()` принимает замыкание с одним аргументом. Обычно его называют `w`. Этот аргумент даёт доступ на чтение и запись к различным подполям регистра, как они определены в файле SVD производителя для данного чипа. Снова: все функции, доступные для `w` этого регистра, в этой периферии, на этом чипе, можно найти в документации tm4c123x. Обратите внимание, что все подполя, которые мы не задаём, получат для нас значения по умолчанию — всё текущее содержимое регистра будет потеряно.

```rust
pwm.ctl.write(|w| w.globalsync0().clear_bit());
```

3.1.3.3. Изменение

Если мы хотим изменить только одно подполе регистра и оставить остальные подполя без изменений, можно использовать функцию `modify`. Она принимает замыкание с двумя аргументами: один для чтения и один для записи. Обычно их называют `r` и `w` соответственно. Аргумент `r` позволяет проверить текущее содержимое регистра, а аргумент `w` — изменить его.

```rust
pwm.ctl.modify(|r, w| w.globalsync0().clear_bit());
```

Функция `modify` наглядно показывает мощь замыканий. В C нам пришлось бы прочитать значение во временную переменную, изменить нужные биты и записать значение обратно. При этом легко допустить ошибку:

```c
uint32_t temp = pwm0.ctl.read();
temp |= PWM0_CTL_GLOBALSYNC0;
pwm0.ctl.write(temp);
uint32_t temp2 = pwm0.enable.read();
temp2 |= PWM0_ENABLE_PWM4EN;
pwm0.enable.write(temp); // Ой! Не та переменная!
```

3.1.4. Использование HAL-крейта

HAL-крейт для чипа обычно работает, реализуя собственный трейт для сырых структур, которые предоставляет PAC. Часто этот трейт определяет функцию `constrain()` для отдельных периферийных блоков или `split()` для таких вещей, как порты GPIO с несколькими выводами. Эта функция забирает владение базовой сырой структурой периферии и возвращает новый объект с более высокоуровневым API. Такой API может также требовать, например, чтобы функция `new` для последовательного порта принимала заимствование структуры `Clock`, которую можно получить только вызовом функции, настраивающей PLL и устанавливающей все частоты тактирования. Таким образом, на этапе компиляции невозможно создать объект последовательного порта, не настроив сначала частоты тактирования, и невозможно, чтобы объект последовательного порта неверно преобразовал скорость передачи в такты. Некоторые крейты даже определяют специальные трейты для состояний, в которых может находиться каждый вывод GPIO, требуя от пользователя перевести вывод в нужное состояние (например, выбрав подходящий режим альтернативной функции), прежде чем передать его в периферию. И всё это без каких-либо затрат во время выполнения!

Посмотрим на пример:

Rust

```rust
#![no_std]
#![no_main]

use panic_halt as _; // обработчик паники

use cortex_m_rt::entry;
use tm4c123x_hal as hal;
use tm4c123x_hal::prelude::*;
use tm4c123x_hal::serial::{NewlineMode, Serial};
use tm4c123x_hal::sysctl;

#[entry]
fn main() -> ! {
    let p = hal::Peripherals::take().unwrap();
    let cp = hal::CorePeripherals::take().unwrap();

    // Оборачиваем структуру SYSCTL в объект с API более высокого уровня
    let mut sc = p.SYSCTL.constrain();
    // Выбираем настройки генератора
    sc.clock_setup.oscillator = sysctl::Oscillator::Main(
        sysctl::CrystalFrequency::_16mhz,
        sysctl::SystemClock::UsePll(sysctl::PllOutputFrequency::_80_00mhz),
    );
    // Настраиваем PLL с этими параметрами
    let clocks = sc.clock_setup.freeze();

    // Оборачиваем структуру GPIO_PORTA в объект с API более высокого уровня.
    // Обратите внимание, что ему нужно заимствовать `sc.power_control`, чтобы
    // автоматически включить периферию GPIO.
```

```
    let mut porta = p.GPIO_PORTA.split(&sc.power_control);

    // Активируем UART.
    let uart = Serial::uart0(
        p.UART0,
        // Выход передачи
        porta
            .pa1
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // Вход приёма
        porta
            .pa0
            .into_af_push_pull::<hal::gpio::AF1>(&mut porta.control),
        // RTS и CTS не нужны
        (),
        (),
        // Скорость передачи
        115200_u32.bps(),
        // Обработка перевода строки на выходе
        NewlineMode::SwapLFtoCRLF,
        // Нам нужны частоты тактирования, чтобы вычислить делители скорости
        передачи
        &clocks,
        // Нужно, чтобы включить периферию UART
        &sc.power_control,
    );

    loop {
        writeln!(uart, "Hello, World!\r\n").unwrap();
    }
}
```

3.2. Semihosting

Semihosting — это механизм, который позволяет встраиваемым устройствам выполнять ввод-вывод на хосте; в основном он используется для вывода сообщений в консоль хоста. Для semihosting нужна отладочная сессия и практически ничего больше (никаких дополнительных проводов!), поэтому им очень удобно пользоваться. Недостаток в том, что он очень медленный: каждая операция записи может занимать несколько миллисекунд в зависимости от аппаратного отладчика (например, ST-Link), который вы используете.

Крейт `cortex-m-semihosting` предоставляет API для операций semihosting на устройствах Cortex-M. Программа ниже — это версия «Hello, world!», использующая semihosting:

```
#![no_main]                                                          Rust
#![no_std]

use panic_halt as _;
```

```
use cortex_m_rt::entry;
use cortex_m_semihosting::hprintln;

#[entry]
fn main() -> ! {
    hprintln!("Hello, world!").unwrap();

    loop {}
}
```

Если вы запустите эту программу на реальном оборудовании, сообщение «Hello, world!» появится в журналах OpenOCD.

```
$ openocd
(..)
Hello, world!
(..)
```

Сначала из GDB нужно включить semihosting в OpenOCD:

```
(gdb) monitor arm semihosting enable
semihosting is enabled
```

QEMU понимает операции semihosting, поэтому программа выше будет работать и с `qemu-system-arm` без запуска отладочной сессии. Обратите внимание, что для включения поддержки semihosting нужно передать QEMU флаг `-semihosting-config`; эти флаги уже включены в файл `.cargo/config.toml` шаблона.

```
$ # эта программа заблокирует терминал
$ cargo run
     Running `qemu-system-arm (..)
Hello, world!
```

Есть также операция semihosting `exit`, которую можно использовать, чтобы завершить процесс QEMU. Важно: **не** используйте `debug::exit` на реальном оборудовании; эта функция может повредить вашу сессию OpenOCD, и вы не сможете отлаживать другие программы, пока не перезапустите её.

```
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
```

```
    let roses = "blue";

    if roses == "red" {
        debug::exit(debug::EXIT_SUCCESS);
    } else {
        debug::exit(debug::EXIT_FAILURE);
    }

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)

$ echo $?
1
```

Ещё один совет: можно настроить поведение при панике так, чтобы оно вызывало `exit(EXIT_FAILURE)`. Это позволит писать `no_std`-тесты, которые проходят выполнение (run-pass), и запускать их на QEMU.

Для удобства у крейта `panic-semihosting` есть feature «exit», который при включении вызывает `exit(EXIT_FAILURE)` после вывода сообщения о панике в stderr хоста.

```
#![no_main]
#![no_std]

use panic_semihosting as _; // features = ["exit"]

use cortex_m_rt::entry;
use cortex_m_semihosting::debug;

#[entry]
fn main() -> ! {
    let roses = "blue";

    assert_eq!(roses, "red");

    loop {}
}
```

```
$ cargo run
     Running `qemu-system-arm (..)
panicked at 'assertion failed: `(left == right)`
  left: `"blue"`,
 right: `"red"`', examples/hello.rs:15:5

$ echo $?
```

```
1
```

ПРИМЕЧАНИЕ: чтобы включить этот feature в `panic-semihosting`, отредактируйте раздел dependencies в вашем `Cargo.toml`, где указан `panic-semihosting`, следующим образом:

```toml
panic-semihosting = { version = "VERSION", features = ["exit"] }
```

где `VERSION` — нужная версия. Подробнее о feature-флагах зависимостей см. раздел `specifying dependencies` в книге Cargo.

3.3. Паника

Паника (panic) — важная часть языка Rust. Встроенные операции, например индексация, проверяются во время выполнения, чтобы гарантировать безопасность памяти. Если выполняется попытка индексации за пределами массива, происходит паника.

В стандартной библиотеке поведение при панике определено: она разматывает стек паникующего потока (unwinding), если только пользователь не выбрал прерывание программы при панике.

Однако в программах без стандартной библиотеки поведение при панике не определено. Его можно выбрать, объявив функцию `#[panic_handler]`. Эта функция должна встречаться в графе зависимостей программы ровно *один* раз и иметь такую сигнатуру: `fn(&PanicInfo) -> !`, где `PanicInfo` — структура, которая содержит информацию о месте возникновения паники.

Поскольку встраиваемые системы бывают самыми разными — от пользовательских устройств до систем, критичных к безопасности (которые не должны падать), — универсального поведения при панике не существует, но есть множество распространённых вариантов. Эти распространённые варианты упакованы в крейты, которые определяют функцию `#[panic_handler]`. Вот некоторые примеры:

- `panic-abort`. При панике выполняется инструкция abort.
- `panic-halt`. При панике программа или текущий поток останавливается, входя в бесконечный цикл.
- `panic-itm`. Сообщение о панике записывается через ITM — периферию, специфичную для ARM Cortex-M.
- `panic-semihosting`. Сообщение о панике передаётся хосту с помощью техники semihosting.

Ещё больше крейтов можно найти на crates.io, выполнив поиск по ключевому слову `panic-handler`.

Программа выбирает одно из этих поведений, просто подключая соответствующий крейт. Тот факт, что поведение при панике задаётся в исходном коде приложения одной строкой, полезен не только как документация, но и позволяет менять поведение при панике в зависимости от профиля сборки. Например:

```rust
#![no_main]
#![no_std]

// профиль dev: удобнее отлаживать паники; можно поставить точку останова на
`rust_begin_unwind`
#[cfg(debug_assertions)]
use panic_halt as _;
```

```
// профиль release: минимизировать размер бинарного файла приложения
#[cfg(not(debug_assertions))]
use panic_abort as _;

// ..
```

В этом примере подключается крейт `panic-halt`, если сборка выполняется с профилем dev (`cargo build`), и `panic-abort`, если с профилем release (`cargo build --release`).

> Форма `use panic_abort as _;` используется, чтобы обработчик паники `panic_abort` попал в итоговый исполняемый файл, а компилятору было понятно, что мы не будем явно использовать ничего из этого крейта. Без переименования `as _` компилятор выдал бы предупреждение о неиспользуемом импорте. Иногда вместо этого встречается `extern crate panic_abort` — более старый стиль, который использовался до издания Rust 2018 и теперь нужен только для «sysroot»-крейтов (поставляемых вместе с самим Rust), таких как `proc_macro`, `alloc`, `std` и `test`.

3.3.1. Пример

Вот пример, который пытается индексировать массив за пределами его длины. Эта операция вызывает панику.

```
#![no_main]                                                    Rust
#![no_std]

use panic_semihosting as _;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    let xs = [0, 1, 2];
    let i = xs.len();
    let _y = xs[i]; // доступ за границами

    loop {}
}
```

В этом примере выбрано поведение `panic-semihosting`, которое выводит сообщение о панике в консоль хоста с помощью semihosting.

```
$ cargo run                                                    Plain Text
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
panicked at 'index out of bounds: the len is 3 but the index is 4', src/
main.rs:12:13
```

Вы можете попробовать поменять поведение на `panic-halt` и убедиться, что в этом случае сообщение не выводится.

3.4. Исключения

Исключения и прерывания — это аппаратный механизм, с помощью которого процессор обрабатывает асинхронные события и фатальные ошибки (например, выполнение недопустимой инструкции). Исключения подразумевают вытеснение (preemption) и используют обработчики исключений — подпрограммы, которые выполняются в ответ на сигнал, вызвавший событие.

Крейт `cortex-m-rt` предоставляет атрибут `exception`, чтобы объявлять обработчики исключений.

```rust
// Обработчик исключения для SysTick (системного таймера)
#[exception]
fn SysTick() {
    // ..
}
```

За исключением атрибута `exception`, обработчики исключений выглядят как обычные функции, но есть одно дополнительное отличие: обработчики `exception` *не могут* быть вызваны программно. Продолжая предыдущий пример, инструкция `SysTick();` приведёт к ошибке компиляции.

Такое поведение, по сути, намеренное и необходимо для одной возможности: `static mut`-переменные, объявленные *внутри* обработчиков `exception`, *безопасно* использовать.

```rust
#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
    // `COUNT` превратилась в тип `&mut u32`, и её безопасно использовать
    *COUNT += 1;
```

Как вы, возможно, знаете, использование `static mut`-переменных внутри функции делает её *не реентерабельной*. Вызывать нереентерабельную функцию, напрямую или косвенно, из более чем одного обработчика исключений или прерываний либо из `main` и одного или нескольких обработчиков исключений или прерываний — неопределённое поведение (UB).

Безопасный Rust никогда не должен приводить к неопределённому поведению, поэтому нереентерабельные функции должны помечаться как `unsafe`. Однако я только что сказал, что обработчики `exception` могут безопасно использовать `static mut`-переменные. Как это возможно? Это возможно потому, что обработчики `exception` *не могут* быть вызваны программно, и, значит, реентерабельность невозможна. Такие обработчики вызываются самим оборудованием, которое, как предполагается, физически не допускает параллельности.

Следовательно, в контексте обработчиков исключений во встраиваемых системах отсутствие параллельных вызовов одного и того же обработчика гарантирует отсутствие проблем с реентерабельностью, даже если обработчик использует изменяемые статические переменные.

В многоядерной системе, где несколько ядер процессора выполняют код параллельно, проблемы реентерабельности снова становятся актуальными, даже внутри обработчиков исключений. Хотя у каждого ядра может быть свой набор обработчиков исключений, всё равно возможны ситуации, когда несколько ядер пытаются выполнить один и тот же обработчик одновременно. Чтобы решить эту проблему в многоядерной среде, внутри обработчиков исключений нужно применять подходящие механизмы синхронизации, гарантирующие,

что доступ к общим ресурсам корректно координируется между ядрами. Обычно для этого используют такие техники, как блокировки, семафоры или атомарные операции, чтобы предотвратить гонки данных и сохранить целостность данных.

> Обратите внимание, что атрибут `exception` преобразует определения статических переменных внутри функции, оборачивая их в блоки `unsafe`, и предоставляет нам новые переменные типа `&mut` с тем же именем. Поэтому мы можем разыменовать ссылку с помощью *, чтобы получить значения переменных, не оборачивая их в блок `unsafe`.

3.4.1. Полный пример

Вот пример, который использует системный таймер, чтобы примерно раз в секунду вызывать исключение `SysTick`. Обработчик исключения `SysTick` подсчитывает, сколько раз он был вызван, в переменной `COUNT`, а затем выводит значение `COUNT` в консоль хоста с помощью semihosting.

> **ПРИМЕЧАНИЕ**: вы можете запустить этот пример на любом устройстве Cortex-M; также его можно запустить на QEMU.

```rust
#![deny(unsafe_code)]
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;

use cortex_m::peripheral::syst::SystClkSource;
use cortex_m_rt::{entry, exception};
use cortex_m_semihosting::{
    debug,
    hio::{self, HostStream},
};

#[entry]
fn main() -> ! {
    let p = cortex_m::Peripherals::take().unwrap();
    let mut syst = p.SYST;

    // настраивает системный таймер так, чтобы он вызывал исключение SysTick каждую секунду
    syst.set_clock_source(SystClkSource::Core);
    // это настроено для LM3S6965, у которого тактовая частота ЦП по умолчанию 12 МГц
    syst.set_reload(12_000_000);
    syst.clear_current();
    syst.enable_counter();
    syst.enable_interrupt();
```

```
    loop {}
}

#[exception]
fn SysTick() {
    static mut COUNT: u32 = 0;
    static mut STDOUT: Option<HostStream> = None;

    *COUNT += 1;

    // Ленивая инициализация
    if STDOUT.is_none() {
        *STDOUT = hio::hstdout().ok();
    }

    if let Some(hstdout) = STDOUT.as_mut() {
        write!(hstdout, "{}", *COUNT).ok();
    }

    // ВАЖНО: уберите этот блок `if`, если запускаете на реальном оборудовании,
    иначе
    // отладчик окажется в несогласованном состоянии
    if *COUNT == 9 {
        // Это завершит процесс QEMU
        debug::exit(debug::EXIT_SUCCESS);
    }
}
```

```
tail -n5 Cargo.toml
```
Shell

```
[dependencies]
cortex-m = "0.5.7"
cortex-m-rt = "0.6.3"
panic-halt = "0.2.0"
cortex-m-semihosting = "0.3.1"
```
TOML

```
$ cargo run --release
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb (..)
123456789
```
Plain Text

Если вы запустите это на отладочной плате Discovery, вывод появится в консоли OpenOCD. Кроме того, программа *не* остановится, когда счётчик дойдёт до 9.

3.4.2. Обработчик исключений по умолчанию

Атрибут `exception` на самом деле *переопределяет* обработчик исключения по умолчанию для конкретного исключения. Если вы не переопределяете обработчик для какого-то исключения, его обработает функция `DefaultHandler`, которая по умолчанию выглядит так:

```rust
fn DefaultHandler() {
    loop {}
}
```

Эта функция предоставляется крейтом `cortex-m-rt` и помечена как `#[no_mangle]`, поэтому на `DefaultHandler` можно поставить точку останова и перехватывать *необработанные* исключения.

`DefaultHandler` можно переопределить с помощью атрибута `exception`:

```rust
#[exception]
fn DefaultHandler(irqn: i16) {
    // пользовательский обработчик по умолчанию
}
```

Аргумент `irqn` указывает, какое исключение обрабатывается. Отрицательное значение означает, что обрабатывается исключение Cortex-M; нулевое или положительное значение означает, что обрабатывается специфичное для устройства исключение, то есть прерывание.

3.4.3. Обработчик Hard Fault

Исключение `HardFault` немного особенное. Оно возникает, когда программа попадает в недопустимое состояние, поэтому его обработчик *не* может вернуть управление, так как это может привести к неопределённому поведению. Кроме того, перед вызовом пользовательского обработчика `HardFault` крейт рантайма выполняет некоторую работу, чтобы облегчить отладку.

В результате обработчик `HardFault` должен иметь такую сигнатуру: `fn(&ExceptionFrame) -> !`. Аргумент обработчика — это указатель на регистры, которые были помещены в стек при исключении. Эти регистры представляют собой снимок состояния процессора в момент возникновения исключения и полезны для диагностики hard fault.

Вот пример, который выполняет недопустимую операцию: чтение из несуществующей области памяти.

> **ПРИМЕЧАНИЕ**: Эта программа не сработает, то есть не упадёт, на QEMU, потому что `qemu-system-arm -machine lm3s6965evb` не проверяет загрузки из памяти и спокойно возвращает `0` при чтении из недопустимой памяти.

```rust
#![no_main]
#![no_std]

use panic_halt as _;

use core::fmt::Write;
use core::ptr;

use cortex_m_rt::{entry, exception, ExceptionFrame};
use cortex_m_semihosting::hio;

#[entry]
```

```
fn main() -> ! {
    // читаем несуществующую область памяти
    unsafe {
        ptr::read_volatile(0x3FFF_0000 as *const u32);
    }

    loop {}
}

#[exception]
fn HardFault(ef: &ExceptionFrame) -> ! {
    if let Ok(mut hstdout) = hio::hstdout() {
        writeln!(hstdout, "{:#?}", ef).ok();
    }

    loop {}
}
```

Обработчик `HardFault` выводит значение `ExceptionFrame`. Если вы запустите этот код, в консоли OpenOCD вы увидите примерно такой результат:

```
$ openocd
(..)
ExceptionFrame {
    r0: 0x3fff0000,
    r1: 0x00000003,
    r2: 0x080032e8,
    r3: 0x00000000,
    r12: 0x00000000,
    lr: 0x080016df,
    pc: 0x080016e2,
    xpsr: 0x61000000,
}
```

Значение `pc` — это значение счётчика команд (Program Counter) в момент исключения; оно указывает на инструкцию, которая вызвала исключение.

Если посмотреть на дизассемблирование программы:

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
(..)
ResetTrampoline:
 8000942:       movw    r0, #0xfffe
 8000946:       movt    r0, #0x3fff
 800094a:       ldr     r0, [r0]
 800094c:       b       #-0x4 <ResetTrampoline+0xa>
```

Значение счётчика команд `0x0800094a` можно найти в дизассемблировании. Вы увидите, что исключение вызвала операция загрузки (`ldr r0, [r0]`). Поле `r0` структуры `ExceptionFrame` покажет, что значение регистра `r0` в тот момент было `0x3fff_fffe`.

3.5. Прерывания

Прерывания отличаются от исключений по многим признакам, но их работа и использование во многом похожи, и обрабатываются они одним и тем же контроллером прерываний. Исключения определяются архитектурой Cortex-M, а прерывания всегда реализуются производителем (и часто даже конкретным чипом) и различаются как по именам, так и по функциональности.

Прерывания дают большую гибкость, которую нужно учитывать при попытке использовать их продвинутым образом. В этой книге мы не будем рассматривать такие случаи, однако полезно помнить следующее:

- У прерываний есть программируемые приоритеты, которые определяют порядок выполнения их обработчиков.
- Прерывания могут вкладываться друг в друга и вытеснять друг друга: выполнение обработчика прерывания может быть прервано другим, более приоритетным прерыванием.
- Обычно причину, вызвавшую прерывание, нужно сбросить, чтобы не допустить бесконечного повторного входа в обработчик прерывания.

Общие шаги инициализации во время выполнения всегда одинаковы:

- Настроить периферию так, чтобы она генерировала запросы прерываний в нужные моменты;
- Задать нужный приоритет обработчика прерывания в контроллере прерываний;
- Разрешить обработчик прерывания в контроллере прерываний.

Подобно исключениям, крейт cortex-m-rt предоставляет атрибут `interrupt` для объявления обработчиков прерываний. Однако этот атрибут доступен только при включённом feature `device`. Кроме того, атрибут не предназначен для прямого использования: попытка сделать это приведёт к ошибке компиляции.

Вместо этого используйте повторно экспортированную версию атрибута interrupt, которую предоставляет крейт устройства (обычно сгенерированный с помощью svd2rust). Это гарантирует, что компилятор может проверить, существует ли прерывание на целевом устройстве. Список доступных прерываний и их позиций в таблице векторов прерываний обычно автоматически генерируется из файла SVD с помощью svd2rust.

```rust
use lm3s6965::interrupt; // Повторно экспортированный атрибут из крейта устройства

// Обработчик прерывания для прерывания Timer2
#[interrupt]
fn TIMER2A() {
    // ..
    // Сбросить причину, по которой сгенерирован запрос прерывания
}
```

Обработчики прерываний выглядят как обычные функции (только без аргументов), аналогично обработчикам исключений. Однако из-за особых соглашений о вызовах другие

части прошивки не могут вызывать их напрямую. Тем не менее можно сгенерировать запрос прерывания программно, чтобы передать управление обработчику прерывания.

Как и в случае с обработчиками исключений, внутри обработчиков прерываний можно объявлять `static mut`-переменные для *безопасного* хранения состояния.

```Rust
#[interrupt]
fn TIMER2A() {
    static mut COUNT: u32 = 0;

    // `COUNT` имеет тип `&mut u32`, и её безопасно использовать
    *COUNT += 1;
}
```

Более подробное описание механизмов, показанных здесь, см. в разделе исключения.

4. Периферия

4.1. Что такое периферия?

У большинства микроконтроллеров есть не только ЦПУ, ОЗУ или флеш-память. Они содержат участки кремния, которые используются для взаимодействия с системами за пределами микроконтроллера, а также для прямого и косвенного взаимодействия с окружающим миром через датчики, контроллеры моторов или пользовательские интерфейсы, такие как дисплей или клавиатура. Эти компоненты в совокупности называются периферией.

Периферия полезна тем, что позволяет разработчику переложить часть обработки на неё и не делать всё программно. Так же, как настольный разработчик переносит обработку графики на видеокарту, встраиваемый разработчик может переложить некоторые задачи на периферию, позволяя ЦПУ заниматься другими важными делами или вообще ничего не делать, чтобы сэкономить энергию.

Если посмотреть на основную плату старого домашнего компьютера 1970-х или 1980-х годов (да и настольные ПК недавнего прошлого не так уж далеки от сегодняшних встраиваемых систем), там можно увидеть:

- процессор;
- микросхему ОЗУ;
- микросхему ПЗУ;
- контроллер ввода-вывода.

Микросхемы ОЗУ и ПЗУ и контроллер ввода-вывода (периферия в этой системе) соединяются с процессором через набор параллельных дорожек, которые называются «шиной» (bus). По этой шине передаётся адресная информация, определяющая, с каким устройством на шине хочет связаться процессор, а также шина данных, по которой передаются собственно данные. В наших встраиваемых микроконтроллерах действуют те же принципы, просто всё упаковано в один кусок кремния.

Однако в отличие от видеокарт, которые обычно имеют программный API вроде Vulkan, Metal или OpenGL, периферия предоставляется нашему микроконтроллеру через аппаратный интерфейс, который отображается в участок памяти.

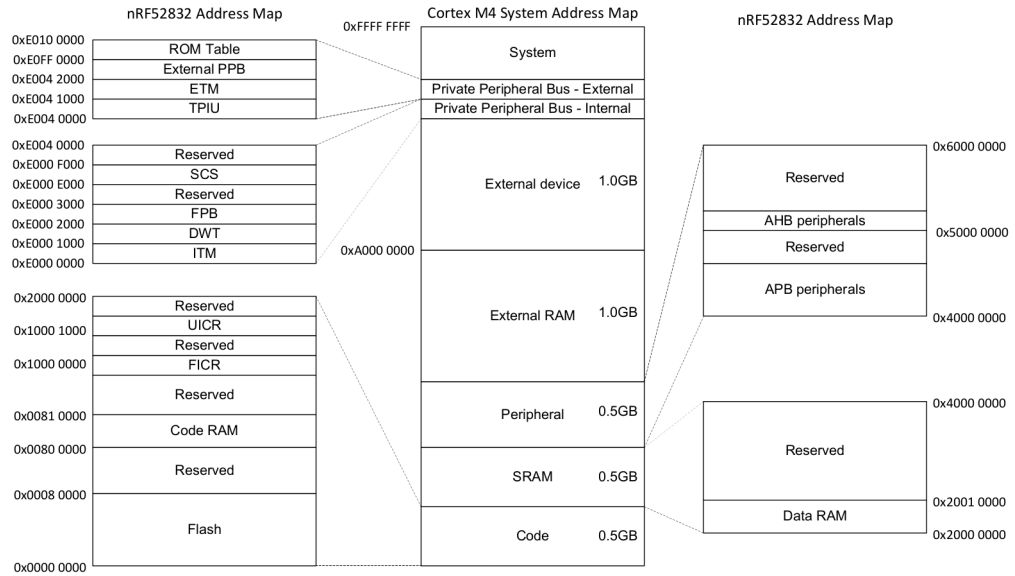
4.2. Линейное адресное пространство и реальная память

На микроконтроллере запись данных по какому-либо произвольному адресу, например `0x4000_0000` или `0x0000_0000`, может быть совершенно допустимым действием.

В настольной системе доступ к памяти жёстко контролирует MMU (Memory Management Unit, блок управления памятью). У этого компонента две основные обязанности: контролировать права доступа к участкам памяти (не давая одному процессу читать или изменять память другого процесса) и переназначать сегменты физической памяти на диапазоны виртуальной памяти, которые используются в программах. У микроконтроллеров обычно нет MMU, и в программах используются только реальные физические адреса.

Хотя 32-битные микроконтроллеры имеют реальное и линейное адресное пространство от `0x0000_0000` до `0xFFFF_FFFF`, они обычно используют для реальной памяти лишь несколько сотен килобайт из этого диапазона. Остаётся значительная часть адресного пространства. В предыдущих главах мы говорили, что ОЗУ расположено по адресу `0x2000_0000`. Если наше ОЗУ имеет размер 64 КиБ (то есть максимальный адрес 0xFFFF), то адреса от `0x2000_0000` до `0x2000_FFFF` соответствуют нашему ОЗУ. Когда мы записываем значение в переменную, которая живёт по адресу `0x2000_1234`, внутри происходит следующее: некоторая логика

определяет старшую часть адреса (в этом примере `0x2000`) и активирует ОЗУ, чтобы оно обработало младшую часть адреса (в данном случае `0x1234`). На Cortex-M флеш-ПЗУ также отображается, например, с адреса `0x0000_0000` до `0x0007_FFFF` (если у нас флеш-ПЗУ на 512 КиБ). Вместо того чтобы игнорировать всё свободное пространство между этими двумя областями, разработчики микроконтроллеров отобразили интерфейсы периферии на определённые адреса памяти. Получается примерно такая картина:



Техническое описание Nordic nRF52832 (pdf)

4.3. Периферия, отображённая в память

На первый взгляд взаимодействие с этой периферией простое: записать правильные данные по правильному адресу. Например, отправка 32-битного слова через последовательный порт может выполняться так же прямо, как запись этого слова по определённому адресу памяти. Периферия последовательного порта затем берёт всё на себя и автоматически отправляет данные.

Настройка этой периферии работает похожим образом. Вместо вызова функции для настройки периферии открывается участок памяти, который служит аппаратным API. Запишите `0x8000_0000` в регистр конфигурации частоты SPI, и SPI-порт будет передавать данные со скоростью 8 Мбит/с. Запишите `0x0200_0000` в тот же адрес — и SPI-порт будет передавать данные со скоростью 125 Кбит/с. Такие регистры конфигурации выглядят примерно так:

31.6.8 FREQUENCY

Address offset: 0x524

SPI frequency

Bit number				31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Id				A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
Reset 0x04000000				**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Id	RW	Field	Value Id	Value	Description
A	RW	FREQUENCY			SPI master data rate
			K125	0x02000000	125 kbps
			K250	0x04000000	250 kbps
			K500	0x08000000	500 kbps
			M1	0x10000000	1 Mbps
			M2	0x20000000	2 Mbps
			M4	0x40000000	4 Mbps
			M8	0x80000000	8 Mbps

Техническое описание Nordic nRF52832 (pdf)

Именно через этот интерфейс осуществляется взаимодействие с аппаратным обеспечением, на каком бы языке ни писали программу: на ассемблере, C или Rust.

4.4. Первая попытка

4.4.1. Регистры

Посмотрим на периферию «SysTick» — простой таймер, который есть в каждом ядре процессора Cortex-M. Обычно такие вещи ищут в техническом описании (data sheet) производителя чипа или в *техническом справочном руководстве* (Technical Reference Manual), но этот пример общий для всех ядер ARM Cortex-M, поэтому заглянем в справочное руководство ARM. Там мы видим четыре регистра:

Смещение	Имя	Описание	Ширина
0x00	SYST_CSR	Регистр управления и состояния	32 бита
0x04	SYST_RVR	Регистр перезагрузочного значения	32 бита
0x08	SYST_CVR	Регистр текущего значения	32 бита
0x0C	SYST_CALIB	Регистр калибровочного значения	32 бита

4.4.2. Подход C

В Rust мы можем представить набор регистров точно так же, как в C, — с помощью `struct`.

```Rust
#[repr(C)]
struct SysTick {
    pub csr: u32,
    pub rvr: u32,
    pub cvr: u32,
    pub calib: u32,
}
```

Атрибут `#[repr(C)]` говорит компилятору Rust расположить эту структуру в памяти так же, как это делает компилятор C. Это очень важно, так как Rust позволяет переупорядочивать поля структуры, а C — нет. Представьте, сколько отладки потребовалось бы, если бы компилятор молча переставлял эти поля! С этим атрибутом у нас есть четыре 32-битных поля, которые соответствуют таблице выше. Но сама по себе эта `struct` бесполезна — нужна переменная.

```Rust
let systick = 0xE000_E010 as *mut SysTick;
let time = unsafe { (*systick).cvr };
```

4.4.3. Летучие (volatile) обращения

У описанного выше подхода есть несколько проблем.

1. Каждый раз, когда мы хотим обратиться к нашей периферии, нужно использовать `unsafe`.
2. У нас нет способа указать, какие регистры доступны только для чтения, а какие — для чтения и записи.
3. Любой фрагмент кода в любом месте программы может обратиться к аппаратуре через эту структуру.
4. Самое важное: это на самом деле не работает…

Проблема в том, что компиляторы умные. Если вы дважды подряд запишете значение в одно и то же место ОЗУ, компилятор может это заметить и просто пропустит первую запись. В C переменные можно пометить как `volatile`, чтобы каждое чтение и запись выполнялись именно так, как задумано. В Rust же помечают не переменную, а *обращения* к ней как volatile.

```Rust
let systick = unsafe { &mut *(0xE000_E010 as *mut SysTick) };
let time = unsafe { core::ptr::read_volatile(&mut systick.cvr) };
```

Итак, мы исправили одну из четырёх проблем, но теперь у нас ещё больше кода с `unsafe`! К счастью, есть сторонний крейт, который может помочь — `volatile_register`.

```Rust
use volatile_register::{RW, RO};

#[repr(C)]
struct SysTick {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

fn get_systick() -> &'static mut SysTick {
    unsafe { &mut *(0xE000_E010 as *mut SysTick) }
}

fn get_time() -> u32 {
    let systick = get_systick();
    systick.cvr.read()
}
```

Теперь volatile-обращения выполняются автоматически через методы `read` и `write`. Запись по-прежнему требует `unsafe`, но справедливости ради скажем, что аппаратура — это набор изменяемого состояния, и у компилятора нет способа узнать, безопасна ли такая запись на самом деле, поэтому это хорошая позиция по умолчанию.

4.4.4. Удобная обёртка на Rust

Нам нужно обернуть эту `struct` в API более высокого уровня, который пользователи смогут безопасно вызывать. Как автор драйвера, мы вручную проверяем, что код с `unsafe` корректен, а затем предоставляем пользователям безопасный API, чтобы им не нужно было об этом беспокоиться (при условии, что они доверяют нам и мы всё сделали правильно!).

Один из вариантов может выглядеть так:

```Rust
use volatile_register::{RW, RO};

pub struct SystemTimer {
    p: &'static mut RegisterBlock
}

#[repr(C)]
```

```rust
struct RegisterBlock {
    pub csr: RW<u32>,
    pub rvr: RW<u32>,
    pub cvr: RW<u32>,
    pub calib: RO<u32>,
}

impl SystemTimer {
    pub fn new() -> SystemTimer {
        SystemTimer {
            p: unsafe { &mut *(0xE000_E010 as *mut RegisterBlock) }
        }
    }

    pub fn get_time(&self) -> u32 {
        self.p.cvr.read()
    }

    pub fn set_reload(&mut self, reload_value: u32) {
        unsafe { self.p.rvr.write(reload_value) }
    }
}

pub fn example_usage() -> String {
    let mut st = SystemTimer::new();
    st.set_reload(0x00FF_FFFF);
    format!("Time is now 0x{:08x}", st.get_time())
}
```

Проблема этого подхода в том, что следующий код для компилятора совершенно допустим:

```rust
fn thread1() {
    let mut st = SystemTimer::new();
    st.set_reload(2000);
}

fn thread2() {
    let mut st = SystemTimer::new();
    st.set_reload(1000);
}
```

Аргумент `&mut self` функции `set_reload` проверяет, что на *эту конкретную* структуру `SystemTimer` нет других ссылок, но они не мешают пользователю создать вторую `SystemTimer`, которая указывает на ту же самую периферию! Код, написанный таким образом, будет работать, если автор достаточно внимателен, чтобы заметить все такие «дублирующиеся» экземпляры драйвера. Но как только код распределяется по нескольким модулям, драйверам, разработчикам и дням, такие ошибки становится всё легче допустить.

4.5. Изменяемое глобальное состояние

К сожалению, аппаратура — это, по сути, не что иное, как изменяемое глобальное состояние, и для разработчика на Rust это может быть очень пугающе. Аппаратура существует независимо от структур нашего кода и может быть изменена в любой момент реальным миром.

4.5.1. Какими должны быть наши правила?

Как нам надёжно взаимодействовать с этой периферией?

1. Всегда использовать `volatile`-методы для чтения и записи в память периферии, так как она может измениться в любой момент.
2. В программе мы должны иметь возможность разделять любое количество доступов только для чтения к этой периферии.
3. Если какой-то части программы нужен доступ на чтение и запись к периферии, она должна владеть единственной ссылкой на эту периферию.

4.5.2. Анализатор заимствований (borrow checker)

Последние два из этих правил звучат подозрительно похоже на то, что уже делает анализатор заимствований!

Представьте, что мы могли бы передавать владение этой периферией или предоставлять неизменяемые или изменяемые ссылки на неё?

Что ж, мы можем, но анализатор заимствований требует, чтобы существовал ровно один экземпляр каждой периферии, чтобы Rust мог корректно с этим работать. К счастью, в аппаратуре действительно есть только один экземпляр любой данной периферии, но как отразить это в структуре нашего кода?

4.6. Синглтоны

> В программной инженерии паттерн «синглтон» — это шаблон проектирования, который ограничивает создание класса одним объектом.
>
> *Википедия: Singleton Pattern*

4.6.1. Почему нельзя просто использовать глобальные переменные?

Мы могли бы сделать всё публичной статической переменной, например так:

```rust
static mut THE_SERIAL_PORT: SerialPort = SerialPort;

fn main() {
    let _ = unsafe {
        THE_SERIAL_PORT.read_speed();
    };
}
```

Но у этого есть несколько проблем. Это изменяемая глобальная переменная, а в Rust работа с такими переменными всегда небезопасна. К тому же эти переменные видны во всей программе, поэтому анализатор заимствований не может помочь отслеживать ссылки и владение ими.

4.6.2. Как это сделать в Rust?

Вместо того чтобы просто делать периферию глобальной переменной, мы можем создать структуру, в данном случае называемую `PERIPHERALS`, которая содержит `Option<T>` для каждой периферии.

```Rust
struct Peripherals {
    serial: Option<SerialPort>,
}
impl Peripherals {
    fn take_serial(&mut self) -> SerialPort {
        let p = replace(&mut self.serial, None);
        p.unwrap()
    }
}
static mut PERIPHERALS: Peripherals = Peripherals {
    serial: Some(SerialPort),
};
```

Эта структура позволяет получить единственный экземпляр нашей периферии. Если мы попытаемся вызвать `take_serial()` больше одного раза, наш код выполнит панику!

```Rust
fn main() {
    let serial_1 = unsafe { PERIPHERALS.take_serial() };
    // Это вызывает панику!
    // let serial_2 = unsafe { PERIPHERALS.take_serial() };
}
```

Хотя работа с этой структурой — `unsafe`, после того как у нас есть содержащийся в ней `SerialPort`, нам больше не нужен ни `unsafe`, ни сама структура `PERIPHERALS`.

У этого есть небольшие накладные расходы во время выполнения: нам нужно обернуть `SerialPort` в option и один раз вызвать `take_serial()`. Однако эта небольшая стоимость в начале позволяет использовать анализатор заимствований во всей остальной программе.

4.6.3. Поддержка в существующих библиотеках

Хотя выше мы создали собственную структуру `Peripherals`, для вашего кода это делать не обязательно. В крейте `cortex_m` есть макрос `singleton!()`, который сделает это за вас.

```Rust
use cortex_m::singleton;

fn main() {
    // Допустимо, если `main` выполняется только один раз
    let x: &'static mut bool =
        singleton!(: bool = false).unwrap();
}
```

документация cortex_m

Кроме того, если вы используете `cortex-m-rtic`, весь процесс определения и получения этой периферии абстрагирован за вас, и вместо этого вы получаете структуру `Peripherals`, которая содержит версию без `Option<T>` для всех элементов, которые вы определили.

```rust
// cortex-m-rtic v0.5.x
#[rtic::app(device = lm3s6965, peripherals = true)]
const APP: () = {
    #[init]
    fn init(cx: init::Context) {
        static mut X: u32 = 0;

        // Периферия Cortex-M
        let core: cortex_m::Peripherals = cx.core;

        // Периферия, специфичная для устройства
        let device: lm3s6965::Peripherals = cx.device;
    }
}
```

4.6.4. Но зачем?

Но как эти синглтоны заметно меняют то, как работает наш код на Rust?

```rust
impl SerialPort {
    const SER_PORT_SPEED_REG: *mut u32 = 0x4000_1000 as _;

    fn read_speed(
        &self // <------ Это действительно очень важно
    ) -> u32 {
        unsafe {
            ptr::read_volatile(Self::SER_PORT_SPEED_REG)
        }
    }
}
```

Здесь важны два фактора:

- Поскольку мы используем синглтон, есть только один способ или одно место, чтобы получить структуру `SerialPort`;
- Чтобы вызвать метод `read_speed()`, нам нужно иметь владение структурой `SerialPort` или ссылку на неё.

Вместе эти два фактора означают, что доступ к аппаратуре возможен только тогда, когда мы надлежащим образом удовлетворили анализатор заимствований, то есть ни в какой момент у нас не может быть нескольких изменяемых ссылок на одну и ту же аппаратуру!

```rust
fn main() {
    // Нет ссылки на `self`! Не сработает.
    // SerialPort::read_speed();

    let serial_1 = unsafe { PERIPHERALS.take_serial() };

    // Можно читать только то, к чему у вас есть доступ
    let _ = serial_1.read_speed();
```

```
}
```

4.6.5. Относитесь к аппаратуре как к данным

Кроме того, поскольку одни ссылки изменяемые, а другие — неизменяемые, становится возможным видеть, может ли функция или метод потенциально изменить состояние аппаратуры.

Например, этой функции разрешено менять настройки аппаратуры:

```rust
fn setup_spi_port(
    spi: &mut SpiPort,
    cs_pin: &mut GpioPin
) -> Result<()> {
    // ...
}
```

А этой — нет:

```rust
fn read_button(gpio: &GpioPin) -> bool {
    // ...
}
```

Это позволяет проверять на этапе **компиляции**, должен ли код вносить изменения в аппаратуру или нет, а не во время выполнения. Стоит заметить, что это обычно работает только в пределах одного приложения, но для bare-metal систем наше ПО компилируется в одно приложение, поэтому это обычно не является ограничением.

5. Статические гарантии

Система типов Rust предотвращает гонки данных на этапе компиляции (см. трейты `Send` и `Sync`). Система типов также может проверять на этапе компиляции и другие свойства, что в некоторых случаях снижает потребность в проверках во время выполнения.

Применительно к встраиваемым программам эти *статические проверки* можно использовать, например, чтобы обеспечить правильную настройку интерфейсов ввода-вывода. Например, можно спроектировать API так, что инициализировать последовательный интерфейс можно только после предварительной настройки выводов, которые будет использовать этот интерфейс.

Также можно статически проверить, что такие операции, как установка вывода в низкий уровень, могут выполняться только над правильно настроенной периферией. Например, попытка изменить выходное состояние вывода, настроенного в режим плавающего входа, вызовет ошибку компиляции.

Кроме того, как мы видели в предыдущей главе, концепцию владения можно применить к периферии, чтобы только определённые части программы могли изменять периферию. Такой *контроль доступа* делает программное обеспечение легче для понимания по сравнению с альтернативой — рассматривать периферию как глобальное изменяемое состояние.

5.1. Программирование на typestate

Концепция typestates описывает кодирование информации о текущем состоянии объекта в типе этого объекта. Хотя это может звучать немного заумно, если вы использовали паттерн Builder в Rust, то вы уже начали использовать программирование на typestate!

Rust

```rust
pub mod foo_module {
    #[derive(Debug)]
    pub struct Foo {
        inner: u32,
    }

    pub struct FooBuilder {
        a: u32,
        b: u32,
    }

    impl FooBuilder {
        pub fn new(starter: u32) -> Self {
            Self {
                a: starter,
                b: starter,
            }
        }

        pub fn double_a(self) -> Self {
            Self {
                a: self.a * 2,
                b: self.b,
```

```
            }
        }

        pub fn into_foo(self) -> Foo {
            Foo {
                inner: self.a + self.b,
            }
        }
    }
}

fn main() {
    let x = foo_module::FooBuilder::new(10)
        .double_a()
        .into_foo();

    println!("{:#?}", x);
}
```

В этом примере нет прямого способа создать объект `Foo`. Нужно создать `FooBuilder` и правильно инициализировать его, прежде чем мы сможем получить нужный объект `Foo`.

Этот минимальный пример кодирует два состояния:
- `FooBuilder`, которое представляет состояние «не настроено» или «настройка в процессе»;
- `Foo`, которое представляет состояние «настроено» или «готово к использованию».

5.1.1. Строгие типы

Поскольку в Rust есть строгая система типов, нет простого способа волшебным образом создать экземпляр `Foo` или превратить `FooBuilder` в `Foo`, не вызвав метод `into_foo()`. Кроме того, вызов метода `into_foo()` поглощает исходную структуру `FooBuilder`, а значит, её нельзя переиспользовать без создания нового экземпляра.

Это позволяет представлять состояния нашей системы в виде типов и включать необходимые действия для переходов между состояниями в методы, которые обменивают один тип на другой. Создав `FooBuilder` и обменяв его на объект `Foo`, мы прошли все шаги простейшего конечного автомата.

5.2. Периферия как конечные автоматы

Периферию микроконтроллера можно рассматривать как набор конечных автоматов. Например, конфигурацию упрощённого вывода GPIO можно представить следующим деревом состояний:
- Отключён (Disabled)
- Включён (Enabled)
  - Настроен как выход (Configured as Output)
    - Выход: высокий уровень (Output: High)
    - Выход: низкий уровень (Output: Low)
  - Настроен как вход (Configured as Input)
    - Вход: высокий импеданс (Input: High Resistance)
    - Вход: подтянут к земле (Input: Pulled Low)
    - Вход: подтянут к питанию (Input: Pulled High)

Если периферия находится в режиме `Disabled`, то чтобы перейти в состояние `Input: High Resistance`, нужно выполнить следующие шаги:
1. Disabled
2. Enabled
3. Configured as Input
4. Input: High Resistance

Если мы хотим перейти из `Input: High Resistance` в `Input: Pulled Low`, нужно выполнить такие шаги:
1. Input: High Resistance
2. Input: Pulled Low

Аналогично, если мы хотим перевести вывод GPIO из состояния `Input: Pulled Low` в `Output: High`, нужно выполнить такие шаги:
1. Input: Pulled Low
2. Configured as Input
3. Configured as Output
4. Output: High

5.2.1. Представление в аппаратуре

Обычно перечисленные выше состояния задаются записью значений в определённые регистры, отображённые на периферию GPIO. Чтобы проиллюстрировать это, определим воображаемый регистр конфигурации GPIO:

Имя	Номер бита(ов)	Значение	Смысл	Примечания
enable	0	0	disabled	Отключает GPIO
		1	enabled	Включает GPIO
direction	1	0	input	Задаёт направление: вход
		1	output	Задаёт направление: выход
input_mode	2..3	00	hi-z	Задаёт вход с высоким сопротивлением
		01	pull-low	Входной вывод подтянут к земле
		10	pull-high	Входной вывод подтянут к питанию
		11	n/a	Недопустимое состояние. Не устанавливать
output_mode	4	0	set-low	Выходной вывод управляется низким уровнем
		1	set-high	Выходной вывод управляется высоким уровнем
input_status	5	x	in-val	0, если вход < 1,5 В; 1, если вход >= 1,5 В

Мы *могли бы* предоставить следующую структуру в Rust для управления этим GPIO:

```rust
/// Интерфейс GPIO
struct GpioConfig {
    /// Структура конфигурации GPIO, сгенерированная svd2rust
    periph: GPIO_CONFIG,
}
```

```
impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) {
        self.periph.modify(|_r, w| {
            w.direction().set_bit(is_output)
        });
    }

    pub fn set_input_mode(&mut self, variant: InputMode) {
        self.periph.modify(|_r, w| {
            w.input_mode().variant(variant)
        });
    }

    pub fn set_output_mode(&mut self, is_high: bool) {
        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });
    }

    pub fn get_input_status(&self) -> bool {
        self.periph.read().input_status().bit_is_set()
    }
}
```

Однако это позволит изменять некоторые регистры бессмысленным образом. Например, что произойдёт, если мы установим поле `output_mode`, когда наш GPIO настроен как вход?

В общем случае использование такой структуры позволило бы прийти к состояниям, не определённым нашим конечным автоматом выше: например, выход, подтянутый к земле, или вход, установленный в высокий уровень. Для некоторой аппаратуры это может не иметь значения. На другой аппаратуре это может привести к неожиданному или неопределённому поведению!

Хотя такой интерфейс удобно писать, он не обеспечивает соблюдения проектных контрактов, заданных нашей аппаратной реализацией.

5.3. Проектные контракты

В предыдущей главе мы написали интерфейс, который *не* обеспечивал соблюдение проектных контрактов. Давайте ещё раз посмотрим на наш воображаемый регистр конфигурации GPIO:

Имя	Номер бита(ов)	Значение	Смысл	Примечания
enable	0	0	disabled	Отключает GPIO
		1	enabled	Включает GPIO
direction	1	0	input	Задаёт направление: вход
		1	output	Задаёт направление: выход
input_mode	2..3	00	hi-z	Задаёт вход с высоким сопротивлением
		01	pull-low	Входной вывод подтянут к земле
		10	pull-high	Входной вывод подтянут к питанию
		11	n/a	Недопустимое состояние. Не устанавливать
output_mode	4	0	set-low	Выходной вывод управляется низким уровнем
		1	set-high	Выходной вывод управляется высоким уровнем
input_status	5	x	in-val	0, если вход < 1,5 В; 1, если вход >= 1,5 В

Если бы вместо этого мы проверяли состояние перед использованием базовой аппаратуры, обеспечивая соблюдение проектных контрактов во время выполнения, мы могли бы написать такой код:

```rust
/// Интерфейс GPIO
struct GpioConfig {
    /// Структура конфигурации GPIO, сгенерированная svd2rust
    periph: GPIO_CONFIG,
}

impl GpioConfig {
    pub fn set_enable(&mut self, is_enabled: bool) {
        self.periph.modify(|_r, w| {
            w.enable().set_bit(is_enabled)
        });
    }

    pub fn set_direction(&mut self, is_output: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Чтобы задать направление, вывод должен быть включён
            return Err(());
        }

        self.periph.modify(|r, w| {
            w.direction().set_bit(is_output)
        });

        Ok(())
```

```
    }

    pub fn set_input_mode(&mut self, variant: InputMode) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Чтобы задать режим входа, вывод должен быть включён
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // Направление должно быть входом
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.input_mode().variant(variant)
        });

        Ok(())
    }

    pub fn set_output_status(&mut self, is_high: bool) -> Result<(), ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Чтобы задать состояние выхода, вывод должен быть включён
            return Err(());
        }

        if self.periph.read().direction().bit_is_clear() {
            // Направление должно быть выходом
            return Err(());
        }

        self.periph.modify(|_r, w| {
            w.output_mode.set_bit(is_high)
        });

        Ok(())
    }

    pub fn get_input_status(&self) -> Result<bool, ()> {
        if self.periph.read().enable().bit_is_clear() {
            // Чтобы получить состояние, вывод должен быть включён
            return Err(());
        }

        if self.periph.read().direction().bit_is_set() {
            // Направление должно быть входом
```

```
            return Err(());
        }

        Ok(self.periph.read().input_status().bit_is_set())
    }
}
```

Поскольку нам нужно обеспечивать соблюдение ограничений аппаратуры, мы в итоге много проверяем во время выполнения. Это тратит время и ресурсы, а работать с таким кодом разработчику гораздо менее приятно.

5.3.1. Состояния в типах (type states)

Но что, если вместо этого мы будем использовать систему типов Rust, чтобы обеспечивать соблюдение правил переходов между состояниями? Посмотрите на этот пример:

```Rust
/// Интерфейс GPIO
struct GpioConfig<ENABLED, DIRECTION, MODE> {
    /// Структура конфигурации GPIO, сгенерированная svd2rust
    periph: GPIO_CONFIG,
    enabled: ENABLED,
    direction: DIRECTION,
    mode: MODE,
}

// Типовые состояния для MODE в GpioConfig
struct Disabled;
struct Enabled;
struct Output;
struct Input;
struct PulledLow;
struct PulledHigh;
struct HighZ;
struct DontCare;

/// Эти функции можно использовать для любого вывода GPIO
impl<EN, DIR, IN_MODE> GpioConfig<EN, DIR, IN_MODE> {
    pub fn into_disabled(self) -> GpioConfig<Disabled, DontCare, DontCare> {
        self.periph.modify(|_r, w| w.enable.disabled());
        GpioConfig {
            periph: self.periph,
            enabled: Disabled,
            direction: DontCare,
            mode: DontCare,
        }
    }

    pub fn into_enabled_input(self) -> GpioConfig<Enabled, Input, HighZ> {
```

```
        self.periph.modify(|_r, w| {
            w.enable.enabled()
             .direction.input()
             .input_mode.high_z()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: HighZ,
        }
    }

    pub fn into_enabled_output(self) -> GpioConfig<Enabled, Output, DontCare> {
        self.periph.modify(|_r, w| {
            w.enable.enabled()
             .direction.output()
             .input_mode.set_high()
        });
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Output,
            mode: DontCare,
        }
    }
}

/// Эту функцию можно использовать для выхода
impl GpioConfig<Enabled, Output, DontCare> {
    pub fn set_bit(&mut self, set_high: bool) {
        self.periph.modify(|_r, w| w.output_mode.set_bit(set_high));
    }
}

/// Эти методы можно использовать для любого включённого входа GPIO
impl<IN_MODE> GpioConfig<Enabled, Input, IN_MODE> {
    pub fn bit_is_set(&self) -> bool {
        self.periph.read().input_status.bit_is_set()
    }

    pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
        self.periph.modify(|_r, w| w.input_mode().high_z());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
```

```
            direction: Input,
            mode: HighZ,
        }
    }

    pub fn into_input_pull_down(self) -> GpioConfig<Enabled, Input, PulledLow> {
        self.periph.modify(|_r, w| w.input_mode().pull_low());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledLow,
        }
    }

    pub fn into_input_pull_up(self) -> GpioConfig<Enabled, Input, PulledHigh> {
        self.periph.modify(|_r, w| w.input_mode().pull_high());
        GpioConfig {
            periph: self.periph,
            enabled: Enabled,
            direction: Input,
            mode: PulledHigh,
        }
    }
}
```

Теперь посмотрим, как выглядел бы код, который это использует:

```Rust
/*
 * Пример 1: от неконфигурированного к входу High-Z
 */
let pin: GpioConfig<Disabled, _, _> = get_gpio();

// Так нельзя, вывод не включён!
// pin.into_input_pull_down();

// Теперь переводим вывод из неконфигурированного состояния во вход High-Z
let input_pin = pin.into_enabled_input();

// Читаем с вывода
let pin_state = input_pin.bit_is_set();

// Так нельзя: у входных выводов нет такого интерфейса!
// input_pin.set_bit(true);

/*
 * Пример 2: вход High-Z → вход с подтяжкой к земле
```

```
 */
let pulled_low = input_pin.into_input_pull_down();
let pin_state = pulled_low.bit_is_set();

/*
 * Пример 3: вход с подтяжкой к земле → выход, установленный в высокий уровень
 */
let output_pin = pulled_low.into_enabled_output();
output_pin.set_bit(true);

// Так нельзя: у выходных выводов нет такого интерфейса!
// output_pin.into_input_pull_down();
```

Это, безусловно, удобный способ хранить состояние вывода. Но зачем делать это именно так? Почему это лучше, чем хранить состояние в виде `enum` внутри нашей структуры `GpioConfig`?

5.3.2. Функциональная безопасность на этапе компиляции

Поскольку мы обеспечиваем соблюдение наших проектных ограничений полностью на этапе компиляции, это не требует никаких затрат во время выполнения. Невозможно задать режим выхода, когда вывод находится в режиме входа. Вместо этого нужно пройти по состояниям: преобразовать вывод в выходной и только после этого задать режим выхода. Поэтому нет штрафа во время выполнения за проверку текущего состояния перед вызовом функции.

Кроме того, поскольку эти состояния обеспечиваются системой типов, у потребителей этого интерфейса не остаётся возможности ошибиться. Если они попытаются выполнить недопустимый переход между состояниями, код не скомпилируется!

5.4. Абстракции с нулевой стоимостью

Typestate — также хороший пример абстракций с нулевой стоимостью (zero cost abstractions): возможности перенести определённое поведение на этап выполнения компилятора или на этап анализа. Эти typestate не содержат никаких реальных данных и используются как маркеры. Поскольку они не содержат данных, у них нет реального представления в памяти во время выполнения:

```Rust
use core::mem::size_of;

let _ = size_of::<Enabled>();    // == 0
let _ = size_of::<Input>();      // == 0
let _ = size_of::<PulledHigh>(); // == 0
let _ = size_of::<GpioConfig<Enabled, Input, PulledHigh>>(); // == 0
```

5.4.1. Типы нулевого размера

```Rust
struct Enabled;
```

Структуры, определённые таким образом, называются типами нулевого размера (Zero Sized Types), поскольку они не содержат никаких реальных данных. Хотя на этапе компиляции эти типы ведут себя как «настоящие» — их можно копировать, перемещать, брать от них ссылки и т. д., — оптимизатор полностью удалит их.

В этом фрагменте кода:

```rust
pub fn into_input_high_z(self) -> GpioConfig<Enabled, Input, HighZ> {
    self.periph.modify(|_r, w| w.input_mode().high_z());
    GpioConfig {
        periph: self.periph,
        enabled: Enabled,
        direction: Input,
        mode: HighZ,
    }
}
```

Возвращаемый `GpioConfig` никогда не существует во время выполнения. Вызов этой функции, как правило, сводится к одной ассемблерной инструкции — записи константного значения регистра в нужное место регистра. Это означает, что построенный нами интерфейс typestate — абстракция с нулевой стоимостью: он не тратит больше процессорного времени, ОЗУ или места в коде на отслеживание состояния `GpioConfig`, и компилируется в тот же машинный код, что и прямой доступ к регистру.

5.4.2. Вложенность

В общем случае такие абстракции можно вкладывать сколько угодно глубоко. Пока все используемые компоненты — типы нулевого размера, вся структура не будет существовать во время выполнения.

Для сложных или глубоко вложенных структур может быть утомительно определять все возможные комбинации состояний. В таких случаях для генерации всех реализаций можно использовать макросы.

6. Переносимость

Во встраиваемых системах переносимость — очень важная тема: каждый производитель и даже каждое семейство одного производителя предлагают разную периферию и возможности, и способы взаимодействия с периферией тоже различаются.

Распространённый способ сгладить эти различия — ввести слой аппаратной абстракции, или **HAL** (Hardware Abstraction Layer).

> Аппаратные абстракции — это наборы программных подпрограмм, которые эмулируют некоторые детали, специфичные для платформы, давая программам прямой доступ к аппаратным ресурсам.
>
> Часто они позволяют программистам писать не зависящие от устройства высокопроизводительные приложения, предоставляя стандартные вызовы операционной системы (OS) для работы с аппаратурой.
>
> *Википедия: Hardware Abstraction Layer*

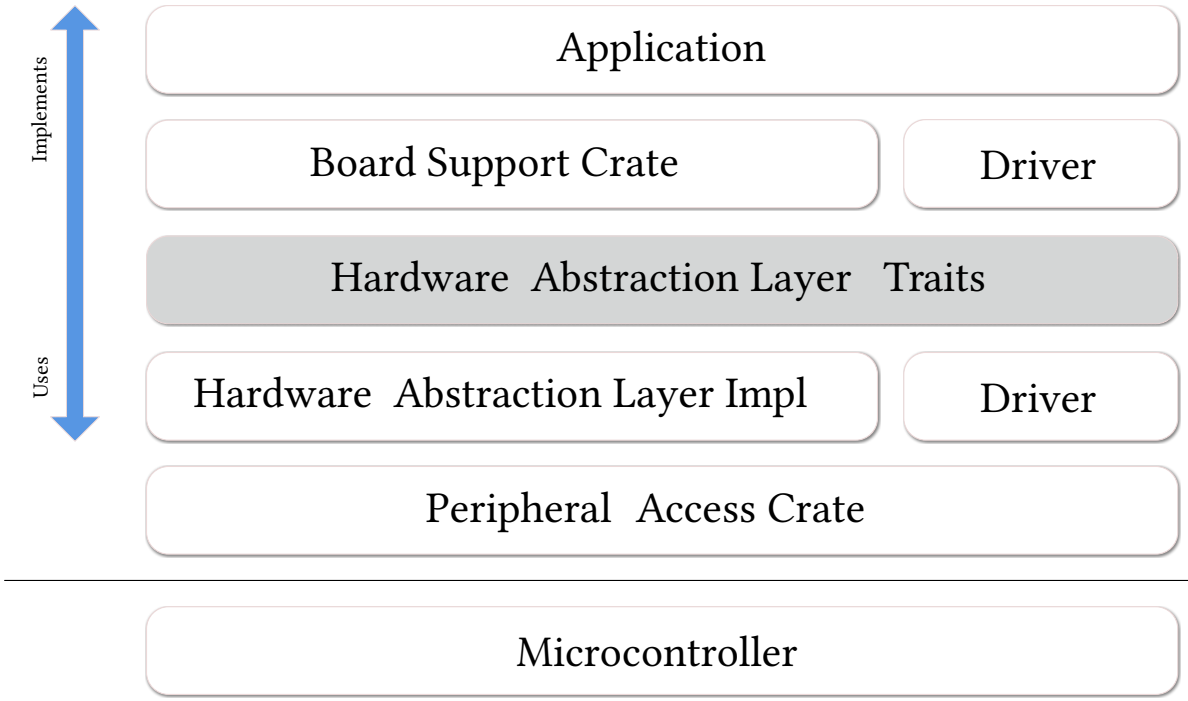
Встраиваемые системы в этом отношении немного особенные: у нас обычно нет операционной системы и устанавливаемого пользователем программного обеспечения, а есть образы прошивки, которые собираются целиком, и ряд других ограничений. Поэтому традиционный подход, описанный в Википедии, потенциально может работать, но, вероятно, не является самым продуктивным способом обеспечить переносимость.

Как же это сделать в Rust? На помощь приходит **embedded-hal**…

6.0.1. Что такое embedded-hal?

Вкратце, это набор трейтов, которые определяют контракты реализации между **реализациями HAL**, **драйверами** и **приложениями (или прошивками)**. Эти контракты включают как возможности (то есть если трейт реализован для определённого типа, то **реализация HAL** предоставляет определённую возможность), так и методы (то есть если вы можете создать тип, реализующий трейт, то гарантированно доступны методы, указанные в трейте).

Типичная схема слоёв может выглядеть так:



Некоторые трейты, определённые в **embedded-hal**:
- GPIO (входные и выходные выводы);
- последовательная передача данных;
- I2C;
- SPI;
- таймеры и отсчёт времени (countdown);
- аналого-цифровое преобразование.

Основная причина, по которой существуют трейты **embedded-hal** и крейты, которые их реализуют и используют, — сдерживание сложности. Если учесть, что приложению может понадобиться реализовать использование периферии в аппаратуре, а также само приложение и, возможно, драйверы для дополнительных аппаратных компонентов, то нетрудно понять, что возможность повторного использования кода очень ограничена. Математически: если **M** — количество реализаций HAL для периферии, а **N** — количество драйверов, то, если заново изобретать колесо для каждого приложения, в итоге получится **M×N** реализаций, тогда как использование **API**, предоставляемого трейтами **embedded-hal**, делает сложность реализации близкой к **M+N**. Конечно, есть и дополнительные преимущества, например меньше проб и ошибок благодаря хорошо определённым и готовым к использованию API.

6.0.2. Пользователи embedded-hal

Как сказано выше, у HAL есть три основных пользователя:

6.0.2.1. Реализация HAL

Реализация HAL обеспечивает взаимодействие между аппаратурой и пользователями трейтов HAL. Типичная реализация состоит из трёх частей:
- одного или нескольких типов, специфичных для аппаратуры;
- функций, которые создают и инициализируют такой тип, часто предоставляя различные параметры конфигурации (скорость, режим работы, используемые выводы и т. д.);
- одной или нескольких реализаций трейтов **embedded-hal** (`impl`) для этого типа.

Такая **реализация HAL** может быть разного вида:
- через низкоуровневый доступ к аппаратуре, например через регистры;

- через операционную систему, например через `sysfs` в Linux;
- через адаптер, например мок-типы для модульного тестирования;
- через драйвер аппаратных адаптеров, например мультиплексора I2C или расширителя GPIO.

6.0.2.2. Драйвер

Драйвер реализует набор специфичной функциональности для внутреннего или внешнего компонента, подключённого к периферии, которая реализует трейты embedded-hal. Типичные примеры таких драйверов — различные датчики (температуры, магнитометр, акселерометр, освещённости), устройства отображения (матрицы светодиодов, ЖК-дисплеи) и исполнительные механизмы (моторы, передатчики).

Драйвер должен инициализироваться экземпляром типа, который реализует определённый трейт embedded-hal; это обеспечивается ограничением трейта (trait bound), и драйвер предоставляет собственный экземпляр типа с набором методов для взаимодействия с управляемым устройством.

6.0.2.3. Приложение

Приложение связывает различные части вместе и обеспечивает достижение нужной функциональности. При переносе между разными системами именно эта часть требует наибольших усилий по адаптации, поскольку приложение должно правильно инициализировать реальное оборудование через реализацию HAL, а инициализация разного оборудования отличается, иногда кардинально. Также часто большую роль играет выбор пользователя: компоненты могут физически подключаться к разным выводам, аппаратные шины иногда требуют внешнего оборудования для согласования конфигурации, а при использовании внутренней периферии приходится выбирать между разными компромиссами (например, доступно несколько таймеров с разными возможностями или периферия конфликтует с другой периферией).

7. Конкурентность

Конкурентность возникает всякий раз, когда разные части программы могут выполняться в разное время или не по порядку. Во встраиваемом контексте сюда относятся:

- обработчики прерываний, которые запускаются каждый раз, когда происходит связанное с ними прерывание;
- различные формы многопоточности, когда ваш микропроцессор регулярно переключается между частями программы;
- а в некоторых системах — многоядерные микропроцессоры, где каждое ядро может независимо выполнять другую часть программы одновременно.

Поскольку многим встраиваемым программам приходится работать с прерываниями, конкурентность так или иначе обычно возникает, и именно здесь могут появляться тонкие и трудные для поиска ошибки. К счастью, Rust предоставляет ряд абстракций и гарантий безопасности, которые помогают писать корректный код.

7.0.1. Без конкурентности

Самая простая конкурентность для встраиваемой программы — её отсутствие: ваше ПО состоит из одного главного цикла, который просто продолжает работать, и никаких прерываний вообще нет. Иногда это прекрасно подходит для решаемой задачи! Обычно ваш цикл считывает входные данные, выполняет обработку и записывает выходные данные.

```rust
#[entry]
fn main() {
    let peripherals = setup_peripherals();
    loop {
        let inputs = read_inputs(&peripherals);
        let outputs = process(inputs);
        write_outputs(&peripherals, outputs);
    }
}
```

Поскольку конкурентности нет, не нужно беспокоиться о совместном использовании данных между частями программы или о синхронизации доступа к периферии. Если вы можете обойтись таким простым подходом, это может стать отличным решением.

7.0.2. Глобальные изменяемые данные

В отличие от не встраиваемого Rust, обычно у нас нет роскоши выделять память в куче и передавать ссылки на эти данные во вновь созданный поток. Вместо этого наши обработчики прерываний могут быть вызваны в любой момент и должны знать, как обращаться к любой общей памяти, которую мы используем. На самом низком уровне это означает, что нам нужна *статически выделенная* изменяемая память, на которую могут ссылаться и обработчик прерывания, и основной код.

В Rust такие переменные `static mut` всегда небезопасно читать или записывать, потому что без особых мер предосторожности вы можете спровоцировать состояние гонки, при котором ваш доступ к переменной прерывается на полпути другим прерыванием, которое также обращается к этой переменной.

Чтобы увидеть, как такое поведение может привести к тонким ошибкам в коде, рассмотрим встраиваемую программу, которая считает фронты входного сигнала в каждом односекундном периоде (частотомер):

```rust
static mut COUNTER: u32 = 0;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // ОПАСНО — на самом деле это небезопасно! Может вызвать гонки данных.
            unsafe { COUNTER += 1 };
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    unsafe { COUNTER = 0; }
}
```

Каждую секунду прерывание таймера сбрасывает счётчик в 0. Тем временем основной цикл непрерывно измеряет сигнал и увеличивает счётчик, когда видит переход от низкого уровня к высокому. Нам пришлось использовать `unsafe`, чтобы обратиться к `COUNTER`, так как это `static mut`, а значит, мы обещаем компилятору, что не вызовем неопределённого поведения. Можете ли вы заметить состояние гонки? Инкремент `COUNTER` *не* гарантированно атомарен: на большинстве встраиваемых платформ он будет разбит на загрузку, затем инкремент, затем сохранение. Если прерывание сработает после загрузки, но до сохранения, сброс в 0 будет проигнорирован после выхода из прерывания — и за этот период мы посчитаем вдвое больше переходов.

7.0.3. Критические секции

Так что же мы можем сделать с гонками данных? Простой подход — использовать *критические секции*, контекст, в котором прерывания запрещены. Оборачивая доступ к `COUNTER` в `main` в критическую секцию, мы можем быть уверены, что прерывание таймера не сработает, пока мы не закончим инкремент `COUNTER`:

```rust
static mut COUNTER: u32 = 0;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // Новая критическая секция обеспечивает синхронизированный доступ к
            COUNTER
```

```
            cortex_m::interrupt::free(|_| {
                unsafe { COUNTER += 1 };
            });
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    unsafe { COUNTER = 0; }
}
```

В этом примере мы используем `cortex_m::interrupt::free`, но на других платформах будут похожие механизмы для выполнения кода в критической секции. Это то же самое, что запретить прерывания, выполнить некоторый код, а затем снова разрешить прерывания.

Обратите внимание, что нам не нужно было помещать критическую секцию внутрь обработчика прерывания, по двум причинам:
- запись 0 в `COUNTER` не может пострадать от гонки, так как мы не читаем его;
- обработчик и так никогда не будет прерван потоком `main`.

Если бы `COUNTER` разделялся между несколькими обработчиками прерываний, которые могут *вытеснять* друг друга, то каждому из них тоже могла бы понадобиться критическая секция.

Это решает нашу непосредственную проблему, но мы всё ещё пишем много небезопасного кода, который нужно тщательно обосновывать, и можем использовать критические секции без необходимости. Поскольку каждая критическая секция временно приостанавливает обработку прерываний, есть связанная с этим цена: немного больший размер кода, большая задержка прерываний и джиттер (прерывания могут обрабатываться дольше, а время до их обработки будет более изменчивым). Является ли это проблемой, зависит от вашей системы, но в общем случае мы хотели бы этого избегать.

Стоит отметить, что хотя критическая секция гарантирует, что прерывания не сработают, она не даёт гарантии эксклюзивности в многоядерных системах! Другое ядро может спокойно обращаться к той же памяти, что и ваше ядро, даже без прерываний. Если вы используете несколько ядер, вам понадобятся более строгие примитивы синхронизации.

7.0.4. Атомарный доступ

На некоторых платформах доступны специальные атомарные инструкции, которые гарантируют атомарность операций чтения-модификации-записи. Для Cortex-M это выглядит так: `thumbv6` (Cortex-M0, Cortex-M0+) предоставляет только атомарные инструкции загрузки и сохранения, тогда как `thumbv7` (Cortex-M3 и выше) предоставляет полные инструкции сравнения и обмена (Compare and Swap, CAS). Эти инструкции CAS дают альтернативу грубому запрету всех прерываний: мы можем попытаться выполнить инкремент, он в большинстве случаев будет успешным, а если его прервали, он автоматически повторит всю операцию инкремента. Эти атомарные операции безопасны даже при нескольких ядрах.

```
use core::sync::atomic::{AtomicUsize, Ordering};                    Rust

static COUNTER: AtomicUsize = AtomicUsize::new(0);
```

```
#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // Используем `fetch_add`, чтобы атомарно прибавить 1 к COUNTER
            COUNTER.fetch_add(1, Ordering::Relaxed);
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Используем `store`, чтобы записать 0 непосредственно в COUNTER
    COUNTER.store(0, Ordering::Relaxed)
}
```

На этот раз `COUNTER` — безопасная переменная `static`. Благодаря типу `AtomicUsize` `COUNTER` можно безопасно изменять и из обработчика прерывания, и из основного потока без запрета прерываний. Когда это возможно, это лучшее решение, но оно может не поддерживаться на вашей платформе.

Замечание о `Ordering`: этот параметр влияет на то, как компилятор и аппаратура могут переупорядочивать инструкции, а также на видимость в кэше. Если предположить, что целевая платформа однопроцессорная, `Relaxed` достаточно и является наиболее эффективным выбором в данном конкретном случае. Более строгий порядок заставит компилятор вставлять барьеры памяти вокруг атомарных операций; в зависимости от того, для чего вы используете атомарные операции, это может быть нужно или нет! Точные детали модели атомарности сложны и лучше описаны в другом месте.

Подробнее про атомарные операции и порядок см. в nomicon.

7.0.5. Абстракции, Send и Sync

Ни одно из описанных выше решений не является особенно удовлетворительным. Они требуют блоков `unsafe`, которые нужно очень тщательно проверять, и они неудобны в использовании. Конечно, в Rust мы можем сделать лучше!

Мы можем обернуть наш счётчик в безопасный интерфейс, который можно безопасно использовать в любом другом месте нашего кода. В этом примере мы используем счётчик на критической секции, но нечто очень похожее можно сделать и с атомарными операциями.

```
use core::cell::UnsafeCell;                                    Rust
use cortex_m::interrupt;

// Наш счётчик — просто обёртка вокруг UnsafeCell<u32>, которая лежит в основе
внутренней
```

```
// изменяемости (interior mutability) в Rust. Используя внутреннюю изменяемость, мы можем
// сделать COUNTER `static`, а не `static mut`, и при этом по-прежнему менять его значение.
struct CSCounter(UnsafeCell<u32>);

const CS_COUNTER_INIT: CSCounter = CSCounter(UnsafeCell::new(0));

impl CSCounter {
    pub fn reset(&self, _cs: &interrupt::CriticalSection) {
        // Требуя передачи CriticalSection, мы знаем, что работаем внутри критической секции,
        // и потому можем уверенно использовать этот блок unsafe (он нужен для вызова UnsafeCell::get).
        unsafe { *self.0.get() = 0 };
    }

    pub fn increment(&self, _cs: &interrupt::CriticalSection) {
        unsafe { *self.0.get() += 1 };
    }
}

// Необходимо, чтобы разрешить static CSCounter. См. объяснение ниже.
unsafe impl Sync for CSCounter {}

// COUNTER больше не `mut`, так как использует внутреннюю изменяемость;
  // поэтому для доступа к нему больше не нужны блоки unsafe.
static COUNTER: CSCounter = CS_COUNTER_INIT;

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            // Тут без unsafe!
            interrupt::free(|cs| COUNTER.increment(cs));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Нам всё же нужно войти в критическую секцию здесь, чтобы получить корректный токен cs,
```

```
    // даже несмотря на то, что мы знаем: никакое другое прерывание не может
    вытеснить это.
    interrupt::free(|cs| COUNTER.reset(cs));

    // Если бы мы действительно хотели, могли бы использовать unsafe-код, чтобы
    создать
    // поддельный CriticalSection и избежать накладных расходов:
    // let cs = unsafe { interrupt::CriticalSection::new() };
}
```

Мы перенесли наш код `unsafe` внутрь тщательно спланированной абстракции, и теперь код приложения не содержит никаких блоков `unsafe`.

Этот дизайн требует, чтобы приложение передавало токен `CriticalSection`: такие токены безопасно создаёт только `interrupt::free`, поэтому, требуя передать один, мы гарантируем, что работаем внутри критической секции, не выполняя блокировку самостоятельно. Эту гарантию компилятор предоставляет статически: с `cs` не связано никаких накладных расходов во время выполнения. Если бы у нас было несколько счётчиков, им всем можно было бы передать один и тот же `cs`, не требуя нескольких вложенных критических секций.

Это также подводит нас к важной теме конкурентности в Rust: трейтам `Send` и `Sync`. Если подытожить книгу по Rust, тип является Send, когда его можно безопасно переместить в другой поток, а Sync — когда им можно безопасно делиться между несколькими потоками. Во встраиваемом контексте мы считаем, что прерывания выполняются в отдельном потоке от кода приложения, поэтому переменные, к которым обращаются и прерывание, и основной код, должны быть Sync.

Для большинства типов в Rust оба этих трейта автоматически выводятся компилятором. Однако поскольку `CSCounter` содержит `UnsafeCell`, он не является Sync, и поэтому мы не могли бы создать `static CSCounter`: переменные `static` *обязаны* быть Sync, так как к ним могут обращаться несколько потоков.

Чтобы сообщить компилятору, что мы позаботились о том, что `CSCounter` действительно безопасно разделять между потоками, мы явно реализуем трейт Sync. Как и в случае предыдущего использования критических секций, это безопасно только на однопроцессорных платформах: при нескольких ядрах потребовались бы дополнительные меры, чтобы обеспечить безопасность.

7.0.6. Мьютексы

Мы создали полезную абстракцию, специфичную для нашей задачи со счётчиком, но для конкурентности существует много распространённых абстракций.

Один из таких *примитивов синхронизации* — мьютекс, сокращение от mutual exclusion (взаимное исключение). Эти конструкции обеспечивают эксклюзивный доступ к переменной, например к нашему счётчику. Поток может попытаться *заблокировать* (или *захватить*) мьютекс, и либо сразу успешно захватывает его, либо блокируется в ожидании захвата, либо получает ошибку о том, что мьютекс заблокировать не удалось. Пока поток удерживает блокировку, ему предоставляется доступ к защищённым данным. Когда поток закончил работу, он *разблокирует* (или *освобождает*) мьютекс, позволяя другому потоку его заблокировать. В Rust разблокировку обычно реализуют с помощью трейта `Drop`, чтобы гарантировать, что она всегда выполняется, когда мьютекс выходит из области видимости.

Использовать мьютекс с обработчиками прерываний может быть непросто: обычно обработчику прерывания нельзя блокироваться, и было бы особенно катастрофично, если бы он заблокировался в ожидании, пока основной поток освободит блокировку, так как тогда возник бы *дедлок* (основной поток никогда не освободит блокировку, потому что выполнение остаётся в обработчике прерывания). Дедлок не считается небезопасным: он возможен даже в безопасном Rust.

Чтобы полностью избежать такого поведения, мы могли бы реализовать мьютекс, для блокировки которого требуется критическая секция, точно так же, как в примере со счётчиком. Пока критическая секция длится столько же, сколько блокировка, мы можем быть уверены в эксклюзивном доступе к обёрнутой переменной, даже без отслеживания состояния блокировки и разблокировки мьютекса.

На самом деле это уже сделано за нас в крейте `cortex_m`! Мы могли бы написать наш счётчик с его помощью:

```
use core::cell::Cell;
use cortex_m::interrupt::Mutex;

static COUNTER: Mutex<Cell<u32>> = Mutex::new(Cell::new(0));

#[entry]
fn main() -> ! {
    set_timer_1hz();
    let mut last_state = false;
    loop {
        let state = read_signal_level();
        if state && !last_state {
            interrupt::free(|cs|
                COUNTER.borrow(cs).set(COUNTER.borrow(cs).get() + 1));
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // Нам всё равно нужно войти в критическую секцию здесь, чтобы удовлетворить
    Mutex.
    interrupt::free(|cs| COUNTER.borrow(cs).set(0));
}
```

Rust

Теперь мы используем `Cell`, который вместе со своим собратом `RefCell` служит для безопасной внутренней изменяемости. Мы уже видели `UnsafeCell` — нижний слой внутренней изменяемости в Rust: он позволяет получить несколько изменяемых ссылок на своё значение, но только с помощью небезопасного кода. `Cell` похож на `UnsafeCell`, но предоставляет безопасный интерфейс: он позволяет только получить копию текущего значения или заменить его, но не получить ссылку, а поскольку он не является Sync, его нельзя разделять между потоками. Эти ограничения делают его безопасным в использовании, но мы

не могли бы использовать его напрямую в переменной `static`, так как `static` должен быть Sync.

Так почему же приведённый выше пример работает? `Mutex<T>` реализует Sync для любого `T`, который является Send, — например, для `Cell`. Он может делать это безопасно, потому что предоставляет доступ к своему содержимому только внутри критической секции. Поэтому мы получаем безопасный счётчик вообще без какого-либо небезопасного кода!

Это отлично подходит для простых типов, таких как `u32` нашего счётчика, но как насчёт более сложных типов, которые не являются Copy? Чрезвычайно распространённый пример во встраиваемом контексте — структура периферии, которая обычно не является Copy. Для этого мы можем обратиться к `RefCell`.

7.0.7. Совместное использование периферии

Крейты устройств, сгенерированные с помощью `svd2rust` и подобных инструментов, обеспечивают безопасный доступ к периферии, гарантируя, что одновременно может существовать только один экземпляр структуры периферии. Это обеспечивает безопасность, но затрудняет доступ к периферии одновременно из основного потока и из обработчика прерывания.

Чтобы безопасно разделить доступ к периферии, мы можем использовать `Mutex`, который мы видели раньше. Нам также понадобится `RefCell`, который использует проверку во время выполнения, чтобы выдавать одну ссылку на периферию за раз. Это создаёт больше накладных расходов, чем простой `Cell`, но поскольку мы выдаём ссылки, а не копии, мы должны быть уверены, что одновременно существует только одна из них.

Наконец, нам также нужно как-то учесть перемещение периферии в разделяемую переменную после её инициализации в основном коде. Для этого мы можем использовать тип `Option`, инициализированный значением `None` и позже установленный в экземпляр периферии.

```rust
use core::cell::RefCell;
use cortex_m::interrupt::{self, Mutex};
use stm32f4::stm32f405;

static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    // Получаем синглтоны периферии и настраиваем её.
    // Этот пример взят из крейта, сгенерированного svd2rust, но
    // большинство крейтов устройств будут похожими.
    let dp = stm32f405::Peripherals::take().unwrap();
    let gpioa = &dp.GPIOA;

    // Какая-то функция конфигурации.
    // Предположим, что она настраивает PA0 как вход, а PA1 как выход.
    configure_gpio(gpioa);

    // Помещаем GPIOA в мьютекс, перемещая его.
```

```
    interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
    // Теперь мы не можем использовать `gpioa` или `dp.GPIOA`, и вместо этого должны
    // обращаться к нему через мьютекс.

    // Обязательно включайте прерывание только после установки MY_GPIO:
    // иначе прерывание может сработать, пока там всё ещё None,
    // и в написанном виде (с `unwrap()`) это вызовет панику.
    set_timer_1hz();
    let mut last_state = false;
    loop {
        // Теперь читаем состояние как цифровой вход через мьютекс
        let state = interrupt::free(|cs| {
            let gpioa = MY_GPIO.borrow(cs).borrow();
            gpioa.as_ref().unwrap().idr.read().idr0().bit_is_set()
        });

        if state && !last_state {
            // Устанавливаем PA1 в высокий уровень, если мы увидели восходящий фронт
            на PA0.
            interrupt::free(|cs| {
                let gpioa = MY_GPIO.borrow(cs).borrow();
                gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
            });
        }
        last_state = state;
    }
}

#[interrupt]
fn timer() {
    // В этот раз в прерывании мы просто сбросим PA0.
    interrupt::free(|cs| {
        // Здесь можно использовать `unwrap()`, потому что мы знаем, что прерывание
        не было включено
        // до того, как был установлен MY_GPIO; иначе нам нужно было бы обработать
        возможность значения None.
        let gpioa = MY_GPIO.borrow(cs).borrow();
        gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().clear_bit());
    });
}
```

Это довольно много для восприятия, поэтому разберём важные строки.

```
static MY_GPIO: Mutex<RefCell<Option<stm32f405::GPIOA>>> =                    Rust
    Mutex::new(RefCell::new(None));
```

Наша разделяемая переменная теперь — `Mutex` вокруг `RefCell`, который содержит `Option`. `Mutex` гарантирует, что доступ есть только во время критической секции, и поэтому делает

переменную Sync, хотя простой `RefCell` не был бы Sync. `RefCell` даёт нам внутреннюю изменяемость через ссылки, которая понадобится для работы с нашим `GPIOA`. `Option` позволяет инициализировать эту переменную пустым значением и только позже действительно переместить в неё нужное значение. Мы не можем обращаться к синглтону периферии статически, только во время выполнения, поэтому это необходимо.

```
interrupt::free(|cs| MY_GPIO.borrow(cs).replace(Some(dp.GPIOA)));
```
Rust

Внутри критической секции мы можем вызвать `borrow()` у мьютекса, который даёт нам ссылку на `RefCell`. Затем мы вызываем `replace()`, чтобы переместить наше новое значение в `RefCell`.

```
interrupt::free(|cs| {
    let gpioa = MY_GPIO.borrow(cs).borrow();
    gpioa.as_ref().unwrap().odr.modify(|_, w| w.odr1().set_bit());
});
```
Rust

Наконец, мы используем `MY_GPIO` безопасным и параллельным образом. Критическая секция не даёт прерыванию сработать, как обычно, и позволяет заимствовать мьютекс. `RefCell` затем даёт нам `&Option<GPIOA>` и отслеживает, как долго он остаётся заимствованным: как только эта ссылка выходит из области видимости, `RefCell` обновляется, показывая, что он больше не заимствован.

Поскольку мы не можем вынуть `GPIOA` из `&Option`, нам нужно преобразовать его в `&Option<&GPIOA>` с помощью `as_ref()`, а затем вызвать `unwrap()`, чтобы получить `&GPIOA`, который позволяет изменять периферию.

Если нам нужна изменяемая ссылка на разделяемый ресурс, вместо этого следует использовать `borrow_mut` и `deref_mut`. Следующий код показывает пример с таймером TIM2.

```
use core::cell::RefCell;
use core::ops::DerefMut;
use cortex_m::interrupt::{self, Mutex};
use cortex_m::asm::wfi;
use stm32f4::stm32f405;

static G_TIM: Mutex<RefCell<Option<Timer<stm32::TIM2>>>> =
    Mutex::new(RefCell::new(None));

#[entry]
fn main() -> ! {
    let mut cp = cm::Peripherals::take().unwrap();
    let dp = stm32f405::Peripherals::take().unwrap();

    // Какая-то функция настройки таймера.
    // Предположим, что она настраивает таймер TIM2, его прерывание NVIC,
    // и наконец запускает таймер.
    let tim = configure_timer_interrupt(&mut cp, dp);

    interrupt::free(|cs| {
        G_TIM.borrow(cs).replace(Some(tim));
```
Rust

```
    });

    loop {
        wfi();
    }
}

#[interrupt]
fn timer() {
    interrupt::free(|cs| {
        if let Some(ref mut tim)) =  G_TIM.borrow(cs).borrow_mut().deref_mut() {
            tim.start(1.hz());
        }
    });
}
```

Уф! Это безопасно, но немного громоздко. Есть ли что-то ещё, что мы можем сделать?

7.0.8. RTIC

Одна из альтернатив — фреймворк RTIC, сокращение от Real Time Interrupt-driven Concurrency (конкурентность реального времени, управляемая прерываниями). Он обеспечивает статические приоритеты и отслеживает доступ к переменным `static mut` («ресурсам»), чтобы статически гарантировать, что разделяемые ресурсы всегда используются безопасно, без накладных расходов на постоянный вход в критические секции и подсчёт ссылок (как в `RefCell`). Это даёт ряд преимуществ, например гарантию отсутствия дедлоков и чрезвычайно малые накладные расходы по времени и памяти.

RTIC поставляется с асинхронным исполнителем, поэтому ваши программные задачи — это функции `async`, в которых можно использовать асинхронные API наряду с обычными синхронными API.

Фреймворк также включает другие возможности, например передачу сообщений, которая уменьшает потребность в явном разделяемом состоянии, и возможность планировать запуск задач на заданное время, что можно использовать для реализации периодических задач. Подробнее смотрите в документации!

7.0.9. Embassy

Embassy — это экосистема библиотек, которая фокусируется на использовании синтаксиса `async` / `await`, встроенного в Rust, для конкурентности. Ядро embassy — её асинхронный исполнитель, который поддерживает большинство распространённых архитектур MCU.

Embassy также придерживается подхода «всё в комплекте» и предлагает множество других компонентов, например:

- Библиотека времени
- Различные библиотеки HAL, которые также обеспечивают поддержку библиотеки времени.
- embassy-sync для примитивов синхронизации

Подробнее см. на сайте и в книге.

7.0.10. Операционные системы реального времени

Другая распространённая модель конкурентности во встраиваемых системах — операционная система реального времени (RTOS). Хотя в Rust их изучено пока меньше, в традиционной встраиваемой разработке они широко используются. Примеры с открытым исходным кодом — FreeRTOS и ChibiOS. Эти RTOS обеспечивают поддержку запуска нескольких потоков приложения, между которыми переключается ЦПУ: либо когда потоки добровольно передают управление (это называется кооперативной многозадачностью), либо на основе регулярного таймера или прерываний (вытесняющая многозадачность). RTOS обычно предоставляют мьютексы и другие примитивы синхронизации и часто взаимодействуют с аппаратными возможностями, такими как движки DMA.

На момент написания мало примеров RTOS на Rust, на которые можно было бы указать, но это интересная область, так что следите за обновлениями!

7.0.11. Несколько ядер

Всё чаще во встраиваемых процессорах бывают два или более ядер, что добавляет ещё один уровень сложности в конкурентность. Все примеры с критической секцией (включая `cortex_m::interrupt::Mutex`) предполагают, что единственный другой поток выполнения — это поток прерываний, но в многоядерной системе это уже не так. Вместо этого нам понадобятся примитивы синхронизации, разработанные для нескольких ядер (также называемые SMP, от symmetric multi-processing — симметричная многопроцессорная обработка).

Обычно они используют атомарные инструкции, которые мы видели ранее, так как система обработки гарантирует сохранение атомарности на всех ядрах.

Подробно рассматривать эти темы в настоящее время выходит за рамки этой книги, но общие шаблоны такие же, как и для однопроцессорного случая.

8. Коллекции

Рано или поздно вам понадобятся динамические структуры данных (они же коллекции) в вашей программе. `std` предоставляет набор распространённых коллекций: `Vec`, `String`, `HashMap` и т. д. Все коллекции, реализованные в `std`, используют глобальный аллокатор динамической памяти (он же куча).

Поскольку `core` по определению не использует выделение памяти, этих реализаций там нет, но их можно найти в крейте `alloc`, который поставляется вместе с компилятором.

Если вам нужны коллекции, реализация с выделением в куче — не единственный вариант. Вы также можете использовать коллекции *фиксированной ёмкости*; одна такая реализация есть в крейте `heapless`.

В этом разделе мы рассмотрим и сравним эти две реализации.

8.0.1. Использование `alloc`

Крейт `alloc` поставляется со стандартным дистрибутивом Rust. Чтобы подключить крейт, можно напрямую использовать `use`, *не* объявляя его как зависимость в файле `Cargo.toml`.

```Rust
#![feature(alloc)]

extern crate alloc;

use alloc::vec::Vec;
```

Чтобы использовать любую коллекцию, сначала нужно с помощью атрибута `global_allocator` объявить глобальный аллокатор, который будет использовать ваша программа. Выбранный аллокатор должен реализовывать трейт `GlobalAlloc`.

Для полноты и чтобы сделать этот раздел максимально самодостаточным, мы реализуем простой аллокатор bump pointer и используем его в качестве глобального аллокатора. Однако мы *настоятельно* советуем использовать в вашей программе проверенный временем аллокатор с crates.io вместо него.

```Rust
// Реализация аллокатора bump pointer

use core::alloc::{GlobalAlloc, Layout};
use core::cell::UnsafeCell;
use core::ptr;

use cortex_m::interrupt;

// Аллокатор bump pointer для *однопроцессорных* систем
struct BumpPointerAlloc {
    head: UnsafeCell<usize>,
    end: usize,
}

unsafe impl Sync for BumpPointerAlloc {}

unsafe impl GlobalAlloc for BumpPointerAlloc {
```

```
    unsafe fn alloc(&self, layout: Layout) -> *mut u8 {
        // `interrupt::free` — это критическая секция, которая делает наш аллокатор
        безопасным
          // для использования из прерываний
        interrupt::free(|_| {
            let head = self.head.get();
            let size = layout.size();
            let align = layout.align();
            let align_mask = !(align - 1);

            // сдвигаем start до следующей границы выравнивания
            let start = (*head + align - 1) & align_mask;

            if start + size > self.end {
                // нулевой указатель сигнализирует об ошибке нехватки памяти (Out Of
                Memory)
                ptr::null_mut()
            } else {
                *head = start + size;
                start as *mut u8
            }
        })
    }

    unsafe fn dealloc(&self, _: *mut u8, _: Layout) {
        // этот аллокатор никогда не освобождает память
    }
}

// Объявление глобального аллокатора памяти
// ВАЖНО: пользователь должен гарантировать, что область памяти `[0x2000_0100,
0x2000_0200]`
// не используется другими частями программы
#[global_allocator]
static HEAP: BumpPointerAlloc = BumpPointerAlloc {
    head: UnsafeCell::new(0x2000_0100),
    end: 0x2000_0200,
};
```

Помимо выбора глобального аллокатора, пользователь также должен определить, как обрабатываются ошибки нехватки памяти (Out Of Memory, OOM), с помощью *нестабильного* атрибута `alloc_error_handler`.

```
#![feature(alloc_error_handler)]                                                Rust

use cortex_m::asm;
```

```
#[alloc_error_handler]
fn on_oom(_layout: Layout) -> ! {
    asm::bkpt();

    loop {}
}
```

Когда всё это настроено, пользователь наконец может использовать коллекции из `alloc`.

```Rust
#[entry]
fn main() -> ! {
    let mut xs = Vec::new();

    xs.push(42);
    assert!(xs.pop(), Some(42));

    loop {
        // ..
    }
}
```

Если вы использовали коллекции из крейта `std`, они покажутся знакомыми, так как реализация точно такая же.

8.0.2. Использование `heapless`

`heapless` не требует никакой настройки, так как его коллекции не зависят от глобального аллокатора памяти. Просто используйте `use` для его коллекций и создавайте их экземпляры:

```Rust
// heapless version: v0.4.x
use heapless::Vec;
use heapless::consts::*;

#[entry]
fn main() -> ! {
    let mut xs: Vec<_, U8> = Vec::new();

    xs.push(42).unwrap();
    assert_eq!(xs.pop(), Some(42));
    loop {}
}
```

Вы заметите два отличия этих коллекций от коллекций из `alloc`.

Во-первых, вам нужно заранее объявить ёмкость коллекции. Коллекции `heapless` никогда не перераспределяются и имеют фиксированную ёмкость; эта ёмкость является частью сигнатуры типа коллекции. В данном случае мы объявили, что у `xs` ёмкость 8 элементов, то есть вектор может содержать не более 8 элементов. Это обозначено `U8` (см. `typenum`) в сигнатуре типа.

Во-вторых, метод `push` и многие другие методы возвращают `Result`. Поскольку коллекции `heapless` имеют фиксированную ёмкость, все операции, которые вставляют элементы

в коллекцию, потенциально могут завершиться неудачей. API отражает эту проблему, возвращая `Result`, который показывает, успешно ли выполнена операция. В отличие от этого, коллекции `alloc` будут перераспределять себя в куче, чтобы увеличить ёмкость.

Начиная с версии v0.4.x все коллекции `heapless` хранят все свои элементы inline. Это означает, что такая операция, как `let x = heapless::Vec::new();`, разместит коллекцию в стеке, но также возможно разместить коллекцию в переменной `static` или даже в куче (`Box<Vec<_, _>>`).

8.0.3. Компромиссы

Имейте это в виду при выборе между коллекциями с выделением в куче, буфер которых может перемещаться при росте, и коллекциями фиксированной ёмкости.

8.0.3.1. Нехватка памяти и обработка ошибок

При выделении в куче нехватка памяти (Out Of Memory) — это всегда возможность, и она может возникнуть в любом месте, где коллекции могут расти: например, любой вызов `alloc::Vec.push` потенциально может привести к OOM. Таким образом, некоторые операции могут *неявно* завершиться неудачей. Некоторые коллекции `alloc` предоставляют методы `try_reserve`, которые позволяют проверить возможные OOM при росте коллекции, но для этого нужно проявлять инициативу и использовать их.

Если вы исключительно используете коллекции `heapless` и не применяете аллокатор памяти ни для чего другого, то OOM-ситуация невозможна. Вместо этого вам придётся обрабатывать случаи, когда коллекции исчерпывают свою ёмкость, в каждом конкретном месте. То есть вам придётся обработать *все* `Result`, возвращаемые методами вроде `Vec.push`.

Ошибки OOM бывает сложнее отлаживать, чем, например, вызов `unwrap` для всех `Result`, возвращаемых `heapless::Vec.push`, потому что наблюдаемое место сбоя может *не* совпадать с местом, где находится причина проблемы. Например, даже `vec.reserve(1)` может вызвать OOM, если аллокатор почти исчерпан, потому что какая-то другая коллекция утекла памятью (утечки памяти возможны в безопасном Rust).

8.0.3.2. Использование памяти

Оценивать использование памяти коллекциями с выделением в куче сложно, потому что ёмкость долгоживущих коллекций может меняться во время выполнения. Некоторые операции могут неявно перераспределять коллекцию, увеличивая её использование памяти, а некоторые коллекции предоставляют методы вроде `shrink_to_fit`, которые потенциально могут уменьшить объём памяти, занятый коллекцией; но в конечном счёте аллокатор решает, действительно ли уменьшать выделение памяти. Кроме того, аллокатору, возможно, придётся бороться с фрагментацией памяти, которая может увеличить *видимое* использование памяти.

С другой стороны, если вы исключительно используете коллекции фиксированной ёмкости, храните большинство из них в переменных `static` и задаёте максимальный размер для стека вызовов, то компоновщик обнаружит, если вы попытаетесь использовать больше памяти, чем физически доступно.

Кроме того, коллекции фиксированной ёмкости, выделенные в стеке, будут учтены флагом `-Z emit-stack-sizes`, а значит, инструменты, которые анализируют использование стека (например, `stack-sizes`), будут включать их в анализ.

Однако коллекции фиксированной ёмкости *не* могут быть уменьшены, из-за чего коэффициент заполнения (отношение размера коллекции к её ёмкости) может быть ниже, чем тот, которого могут достичь перемещаемые коллекции.

8.0.3.3. Наихудшее время выполнения (WCET)

Если вы создаёте чувствительные ко времени или приложения жёсткого реального времени, то вас, возможно, очень волнует наихудшее время выполнения (worst case execution time, WCET) разных частей вашей программы.

Коллекции `alloc` могут перераспределяться, поэтому WCET операций, которые могут увеличивать коллекцию, также будет включать время, которое занимает перераспределение. Оно, в свою очередь, зависит от *времени выполнения* ёмкости коллекции. Из-за этого трудно определить WCET, например, операции `alloc::Vec.push`, так как она зависит и от используемого аллокатора, и от его ёмкости во время выполнения.

С другой стороны, коллекции фиксированной ёмкости никогда не перераспределяются, поэтому все операции имеют предсказуемое время выполнения. Например, `heapless::Vec.push` выполняется за константное время.

8.0.3.4. Простота использования

`alloc` требует настройки глобального аллокатора, тогда как `heapless` — нет. Однако `heapless` требует, чтобы вы выбрали ёмкость каждой создаваемой коллекции.

API `alloc` будет знаком практически каждому разработчику на Rust. API `heapless` старается максимально походить на API `alloc`, но никогда не будет абсолютно таким же из-за явной обработки ошибок — некоторым разработчикам явная обработка ошибок может показаться избыточной или слишком громоздкой.

9. Паттерны проектирования

В этой главе собраны различные полезные паттерны проектирования для встраиваемого Rust.

9.1. Паттерны проектирования HAL

Это набор общих и рекомендуемых паттернов для написания слоёв аппаратной абстракции (HAL) для микроконтроллеров на Rust. Эти паттерны предназначены для использования наряду с существующими Rust API Guidelines при написании HAL для микроконтроллеров.
Чек-лист

- Именование
- Совместимость
- Предсказуемость
- GPIO

9.1.1. Чек-лист паттернов проектирования HAL

- **Именование** (*крейт соответствует соглашениям Rust об именовании*)
  - ☐ Крейт назван надлежащим образом (C-CRATE-NAME)
- **Совместимость** (*крейт корректно взаимодействует с другой библиотечной функциональностью*)
  - ☐ Типы-обёртки предоставляют метод деструктора (C-FREE)
  - ☐ HAL реэкспортируют свой крейт доступа к регистрам (C-REEXPORT-PAC)
  - ☐ Типы реализуют трейты `embedded-hal` (C-HAL-TRAITS)
- **Предсказуемость** (*крейт делает код понятным, а его поведение соответствует внешнему виду кода*)
  - ☐ Используются конструкторы вместо extension-трейтов (C-CTOR)
- **Интерфейсы GPIO** (*интерфейсы GPIO следуют общему шаблону*)
  - ☐ Типы выводов по умолчанию имеют нулевой размер (C-ZST-PIN)
  - ☐ Типы выводов предоставляют методы стирания типа вывода и порта (C-ERASED-PIN)
  - ☐ Состояние вывода должно кодироваться как параметры типа (C-PIN-STATE)

9.1.2. Именование

9.1.2.1. Крейт назван надлежащим образом (C-CRATE-NAME)

Крейты HAL должны называться по имени микросхемы или семейства микросхем, которые они призваны поддерживать. Их название должно заканчиваться на `-hal`, чтобы отличать их от крейтов доступа к регистрам. Название не должно содержать символов подчёркивания (вместо них используйте дефисы).

9.1.3. Совместимость

9.1.3.1. Типы-обёртки предоставляют метод деструктора (C-FREE)

Любой не-`Copy` тип-обёртка, предоставляемый HAL, должен предоставлять метод `free`, который поглощает обёртку и возвращает сырую периферию (и, возможно, другие объекты), из которых она была создана.

Метод должен при необходимости выключать и сбрасывать периферию. Вызов `new` с сырой периферией, возвращённой `free`, не должен завершаться с ошибкой из-за неожиданного состояния периферии.

Если типу HAL требуется создать другие не-`Copy` объекты (например, выводы ввода-вывода), все такие объекты также должны освобождаться и возвращаться через `free`. В этом случае `free` должен возвращать кортеж.

Например:

```Rust
# pub struct TIMER0;
pub struct Timer(TIMER0);

impl Timer {
    pub fn new(periph: TIMER0) -> Self {
        Self(periph)
    }

    pub fn free(self) -> TIMER0 {
        self.0
    }
}
```

9.1.3.2. HAL реэкспортируют свой крейт доступа к регистрам (C-REEXPORT-PAC)

HAL можно писать поверх PAC, сгенерированных svd2rust, или поверх других крейтов, которые предоставляют прямой доступ к регистрам. HAL всегда должны реэкспортировать в корне своего крейта крейт доступа к регистрам, на котором они основаны.

PAC следует реэкспортировать под именем `pac` независимо от фактического имени крейта, так как название HAL уже должно ясно показывать, к какому PAC выполняется доступ.

9.1.3.3. Типы реализуют трейты `embedded-hal` (C-HAL-TRAITS)

Типы, предоставляемые HAL, должны реализовывать все применимые трейты, которые предоставляет крейт `embedded-hal`.

Для одного типа может быть реализовано несколько трейтов.

9.1.4. Предсказуемость

9.1.4.1. Конструкторы используются вместо extension-трейтов (C-CTOR)

Вся периферия, к которой HAL добавляет функциональность, должна оборачиваться в новый тип, даже если для этой функциональности не требуются дополнительные поля.

Следует избегать extension-трейтов, реализованных для сырой периферии.

9.1.4.2. Методы помечаются `#[inline]` там, где это уместно (C-INLINE)

Компилятор Rust по умолчанию не выполняет полную инлайнизацию через границы крейтов. Поскольку встраиваемые приложения чувствительны к неожиданному росту размера кода, `#[inline]` следует использовать, чтобы направлять компилятор следующим образом:

- Все «маленькие» функции следует помечать `#[inline]`. Что считать «маленькой» функцией — вопрос субъективный, но, как правило, к маленьким относятся все функции, которые, как ожидается, компилируются в последовательности из одной или нескольких инструкций.
- Функции, которые с большой вероятностью получат константные значения в качестве параметров, следует помечать как `#[inline]`. Это позволяет компилятору вычислить даже сложную логику инициализации на этапе компиляции, если входные данные функции известны.

9.1.5. Рекомендации для интерфейсов GPIO

9.1.5.1. Типы выводов по умолчанию имеют нулевой размер (C-ZST-PIN)

Интерфейсы GPIO, предоставляемые HAL, должны предоставлять выделенные типы нулевого размера для каждого вывода на каждом интерфейсе или порту. Это даёт абстракцию GPIO с нулевой стоимостью, когда все назначения выводов известны статически.

Каждый интерфейс или порт GPIO должен реализовывать метод `split`, который возвращает структуру со всеми выводами.

Пример:

```rust
pub struct PA0;
pub struct PA1;
// ...

pub struct PortA;

impl PortA {
    pub fn split(self) -> PortAPins {
        PortAPins {
            pa0: PA0,
            pa1: PA1,
            // ...
        }
    }
}

pub struct PortAPins {
    pub pa0: PA0,
    pub pa1: PA1,
    // ...
}
```

9.1.5.2. Типы выводов предоставляют методы стирания вывода и порта (C-ERASED-PIN)

Выводы должны предоставлять методы стирания типа, которые переносят их свойства из времени компиляции во время выполнения и дают больше гибкости в приложениях.

Пример:

```rust
/// Порт A, вывод 0.
pub struct PA0;

impl PA0 {
    pub fn erase_pin(self) -> PA {
        PA { pin: 0 }
    }
}
```

```rust
/// Вывод порта A.
pub struct PA {
    /// Номер вывода.
    pin: u8,
}

impl PA {
    pub fn erase_port(self) -> Pin {
        Pin {
            port: Port::A,
            pin: self.pin,
        }
    }
}

pub struct Pin {
    port: Port,
    pin: u8,
    // (эти поля можно упаковать, чтобы уменьшить объём памяти)
}

enum Port {
    A,
    B,
    C,
    D,
}
```

9.1.5.3. Состояние вывода должно кодироваться как параметры типа (C-PIN-STATE)

Выводы могут быть настроены как входы или выходы с разными характеристиками в зависимости от микросхемы или семейства. Это состояние должно кодироваться в системе типов, чтобы предотвратить использование выводов в неправильных состояниях.

Дополнительное, специфичное для микросхемы состояние (например, нагрузочная способность выхода) также можно кодировать таким образом, используя дополнительные параметры типа.

Методы изменения состояния вывода должны предоставляться в виде методов `into_input` и `into_output`.

Кроме того, должны предоставляться методы `with_{input,output}_state`, которые временно переконфигурируют вывод в другое состояние, не перемещая его.

Для каждого типа вывода должны предоставляться следующие методы (то есть и стёртые, и нестёртые типы выводов должны предоставлять один и тот же API):

- `pub fn into_input<N: InputState>(self, input: N) -> Pin<N>`
- `pub fn into_output<N: OutputState>(self, output: N) -> Pin<N>`
-
```rust
pub fn with_input_state<N: InputState, R>(
    &mut self,
```
Rust

```
    input: N,
    f: impl FnOnce(&mut PA1<N>) -> R,
) -> R
```

- ```rust
  pub fn with_output_state<N: OutputState, R>(
      &mut self,
      output: N,
      f: impl FnOnce(&mut PA1<N>) -> R,
  ) -> R
  ```

Состояние вывода должно ограничиваться запечатанными трейтами (sealed traits). Пользователям HAL не должно требоваться добавлять собственные состояния. Трейты могут предоставлять специфичные для HAL методы, необходимые для реализации API состояний вывода.

Пример:

```rust
# use std::marker::PhantomData;
mod sealed {
    pub trait Sealed {}
}

pub trait PinState: sealed::Sealed {}
pub trait OutputState: sealed::Sealed {}
pub trait InputState: sealed::Sealed {
    // ...
}

pub struct Output<S: OutputState> {
    _p: PhantomData<S>,
}

impl<S: OutputState> PinState for Output<S> {}
impl<S: OutputState> sealed::Sealed for Output<S> {}

pub struct PushPull;
pub struct OpenDrain;

impl OutputState for PushPull {}
impl OutputState for OpenDrain {}
impl sealed::Sealed for PushPull {}
impl sealed::Sealed for OpenDrain {}

pub struct Input<S: InputState> {
    _p: PhantomData<S>,
}

impl<S: InputState> PinState for Input<S> {}
```
```

```
impl<S: InputState> sealed::Sealed for Input<S> {}

pub struct Floating;
pub struct PullUp;
pub struct PullDown;

impl InputState for Floating {}
impl InputState for PullUp {}
impl InputState for PullDown {}
impl sealed::Sealed for Floating {}
impl sealed::Sealed for PullUp {}
impl sealed::Sealed for PullDown {}

pub struct PA1<S: PinState> {
    _p: PhantomData<S>,
}

impl<S: PinState> PA1<S> {
    pub fn into_input<N: InputState>(self, input: N) -> PA1<Input<N>> {
        todo!()
    }

    pub fn into_output<N: OutputState>(self, output: N) -> PA1<Output<N>> {
        todo!()
    }

    pub fn with_input_state<N: InputState, R>(
        &mut self,
        input: N,
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
    }

    pub fn with_output_state<N: OutputState, R>(
        &mut self,
        output: N,
        f: impl FnOnce(&mut PA1<N>) -> R,
    ) -> R {
        todo!()
    }
}

// То же самое для `PA` и `Pin`, а также других типов выводов.
```

10. Советы для разработчиков встраиваемого C

В этой главе собраны различные советы, которые могут пригодиться опытным разработчикам встраиваемого C, которые хотят начать писать на Rust. В особенности здесь подчёркивается, чем на Rust отличаются вещи, к которым вы, возможно, уже привыкли в C.

10.0.1. Препроцессор

Во встраиваемом C очень распространено использование препроцессора для самых разных целей, например:

- выбор блоков кода на этапе компиляции с помощью `#ifdef`;
- размеры массивов и вычисления на этапе компиляции;
- макросы для упрощения типовых конструкций (чтобы избежать накладных расходов на вызов функции).

В Rust препроцессора нет, поэтому многие из этих задач решаются иначе. В остальной части этого раздела мы рассмотрим различные альтернативы использованию препроцессора.

10.0.1.1. Выбор кода на этапе компиляции

Ближайший аналог `#ifdef ... #endif` в Rust — это feature-флаги Cargo (features). Они немного формальнее, чем препроцессор C: все возможные feature явно перечисляются для каждого крейта и могут быть только включены или выключены. Feature включаются, когда вы указываете крейт как зависимость, и они аддитивны: если какой-либо крейт в дереве зависимостей включает feature другого крейта, этот feature будет включён для всех пользователей этого крейта.

Например, у вас может быть крейт, который предоставляет библиотеку примитивов обработки сигналов. Каждый из них может требовать дополнительного времени на компиляцию или объявлять большую таблицу констант, которой вы хотели бы избежать. Вы можете объявить feature Cargo для каждого компонента в своём `Cargo.toml`:

```toml
[features]
FIR = []
IIR = []
```

Затем в коде используйте `#[cfg(feature="FIR")]`, чтобы управлять тем, что включается.

```rust
/// В вашем верхнеуровневом lib.rs

#[cfg(feature="FIR")]
pub mod fir;

#[cfg(feature="IIR")]
pub mod iir;
```

Аналогичным образом можно включать блоки кода, только если feature *не* включён, или при любом сочетании включённых и выключенных feature.

Кроме того, Rust предоставляет ряд автоматически задаваемых условий, которые можно использовать, например `target_arch`, чтобы выбирать разный код в зависимости от архитектуры. Полные подробности о поддержке условной компиляции см. в главе условная компиляция справочника Rust.

Условная компиляция применяется только к следующему оператору или блоку. Если блок нельзя использовать в текущей области видимости, атрибут `cfg` придётся указывать несколько раз. Стоит отметить, что чаще всего лучше просто включить весь код и позволить компилятору удалить мёртвый код при оптимизации: это проще для вас и ваших пользователей, и, как правило, компилятор хорошо справляется с удалением неиспользуемого кода.

10.0.1.2. Размеры и вычисления на этапе компиляции

Rust поддерживает `const fn` — функции, которые гарантированно вычисляются на этапе компиляции и поэтому могут использоваться там, где требуются константы, например для размера массивов. Это можно сочетать с описанными выше feature, например:

```rust
const fn array_size() -> usize {
    #[cfg(feature="use_more_ram")]
    { 1024 }
    #[cfg(not(feature="use_more_ram"))]
    { 128 }
}

static BUF: [u32; array_size()] = [0u32; array_size()];
```

Эти возможности появились в стабильном Rust начиная с версии 1.31, поэтому документации по ним пока мало. Функциональность, доступная для `const fn`, на момент написания также очень ограничена; ожидается, что в будущих версиях Rust круг допустимого в `const fn` расширится.

10.0.1.3. Макросы

Rust предоставляет чрезвычайно мощную систему макросов. Препроцессор C работает почти непосредственно с текстом исходного кода, тогда как система макросов Rust работает на более высоком уровне. Существует два вида макросов Rust: *макросы по примеру* (macros by example) и *процедурные макросы*. Первые проще и распространённее: они выглядят как вызовы функций и могут раскрываться в целое выражение, оператор, элемент или шаблон. Процедурные макросы сложнее, но позволяют очень мощно расширять язык Rust: они могут преобразовывать произвольный синтаксис Rust в новый синтаксис Rust.

В общем случае там, где вы могли бы использовать макрос препроцессора C, стоит проверить, справится ли с задачей макрос по примеру. Такие макросы можно определить в своём крейте, легко использовать внутри собственного крейта или экспортировать для других пользователей. Имейте в виду, что, поскольку они должны раскрываться в целые выражения, операторы, элементы или шаблоны, некоторые сценарии использования макросов препроцессора C не будут работать; например, макрос, который раскрывается в часть имени переменной, или неполный набор элементов в списке.

Как и в случае с feature-флагами Cargo, стоит подумать, нужен ли вам вообще макрос. Во многих случаях обычная функция проще для понимания и будет встроена (inlined) в тот же код, что и макрос. Атрибуты `#[inline]` и `#[inline(always)]` дают дополнительный контроль над этим процессом, но и здесь нужна осторожность: компилятор автоматически встраивает функции из того же крейта там, где это уместно, поэтому принудительное встраивание не к месту может привести к снижению производительности.

Подробно объяснять всю систему макросов Rust выходит за рамки этой страницы с советами, поэтому мы рекомендуем обратиться к документации Rust за полными сведениями.

10.0.2. Система сборки

Большинство крейтов Rust собираются с помощью Cargo (хотя это не обязательно). Cargo решает многие сложные проблемы традиционных систем сборки. Однако вы можете захотеть настроить процесс сборки. Для этого Cargo предоставляет скрипты `build.rs`. Это скрипты на Rust, которые при необходимости могут взаимодействовать с системой сборки Cargo.

Типичные сценарии использования скриптов сборки:

- предоставление информации, известной на этапе сборки, например статическое встраивание даты сборки или хеша коммита Git в исполняемый файл;
- генерация скриптов компоновщика на этапе сборки в зависимости от выбранных feature-флагов или другой логики;
- изменение конфигурации сборки Cargo;
- добавление дополнительных статических библиотек для компоновки.

На данный момент постсборочные скрипты не поддерживаются — то есть те, которые традиционно использовались для таких задач, как автоматическое создание бинарных файлов из объектных файлов сборки или вывод информации о сборке.

10.0.2.1. Кросс-компиляция

Использование Cargo в качестве системы сборки также упрощает кросс-компиляцию. В большинстве случаев достаточно указать Cargo `--target thumbv6m-none-eabi` и найти подходящий исполняемый файл в `target/thumbv6m-none-eabi/debug/myapp`.

Для платформ, которые Rust не поддерживает нативно, вам самостоятельно понадобится собрать `libcore` для этой целевой платформы. Для таких платформ можно использовать Xargo как замену Cargo, которая автоматически собирает `libcore` за вас.

10.0.3. Итераторы вместо доступа по индексу

В C вы, вероятно, привыкли обращаться к массивам напрямую по индексу:

```c
int16_t arr[16];
int i;
for(i=0; i<sizeof(arr)/sizeof(arr[0]); i++) {
    process(arr[i]);
}
```

В Rust это антипаттерн: доступ по индексу может быть медленнее (так как требует проверки границ) и может помешать различным оптимизациям компилятора. Это важное различие, поэтому стоит повторить: Rust проверяет выход за границы при ручной индексации массива, чтобы гарантировать безопасность памяти, тогда как C спокойно позволит индексировать за пределами массива.

Вместо этого используйте итераторы:

```rust
let arr = [0u16; 16];
for element in arr.iter() {
    process(*element);
}
```

Итераторы предоставляют мощный набор функциональности, который в C пришлось бы реализовывать вручную: цепочки, объединение (zip), нумерацию (enumerate), поиск

минимума или максимума, суммирование и многое другое. Методы итераторов также можно объединять в цепочки, что даёт очень читаемый код обработки данных.

Подробнее см. Итераторы в книге и документацию по итераторам.

10.0.4. Ссылки и указатели

В Rust указатели (называемые *сырыми указателями*) существуют, но используются только в особых случаях, поскольку разыменование их всегда считается `unsafe` — Rust не может обеспечить свои обычные гарантии относительно того, что находится за указателем.

В большинстве случаев вместо них мы используем *ссылки*, обозначаемые символом &, или *изменяемые ссылки*, обозначаемые `&mut`. Ссылки ведут себя похоже на указатели в том смысле, что их можно разыменовать, чтобы получить доступ к базовым значениям, но они являются ключевой частью системы владения Rust: Rust строго следит за тем, что в любой момент времени для одного и того же значения может существовать только одна изменяемая ссылка *или* несколько неизменяемых ссылок.

На практике это означает, что нужно внимательнее относиться к тому, нужен ли вам изменяемый доступ к данным: тогда как в C по умолчанию всё изменяемо и нужно явно указывать `const`, в Rust верно обратное.

Одна из ситуаций, в которых вы всё же можете использовать сырые указатели, — прямое взаимодействие с аппаратурой (например, запись указателя на буфер в регистр периферии DMA). Кроме того, они используются «под капотом» всех крейтов доступа к периферии, чтобы позволить читать и записывать регистры, отображённые в память.

10.0.5. Летучий (volatile) доступ

В C отдельные переменные можно пометить как `volatile`, что указывает компилятору, что значение переменной может измениться между обращениями. Volatile-переменные обычно используются во встраиваемом контексте для регистров, отображённых в память.

В Rust вместо пометки переменной как `volatile` мы используем специальные методы для volatile-доступа: `core::ptr::read_volatile` и `core::ptr::write_volatile`. Эти методы принимают `*const T` или `*mut T` (*сырые указатели*, о которых говорилось выше) и выполняют volatile-чтение или запись.

Например, в C можно написать:

```c
volatile bool signalled = false;

void ISR() {
    // Сигнализируем, что прерывание произошло
    signalled = true;
}

void driver() {
    while(true) {
        // Спим, пока не получим сигнал
        while(!signalled) { WFI(); }
        // Сбрасываем индикатор сигнала
        signalled = false;
        // Выполняем задачу, которая ждала прерывания
```

```
        run_task();
    }
}
```

Эквивалент на Rust использовал бы volatile-методы при каждом обращении:

```rust
static mut SIGNALLED: bool = false;

#[interrupt]
fn ISR() {
    // Сигнализируем, что прерывание произошло
    // (В реальном коде стоит рассмотреть примитив более высокого уровня,
    //  например атомарный тип.)
    unsafe { core::ptr::write_volatile(&mut SIGNALLED, true) };
}

fn driver() {
    loop {
        // Спим, пока не получим сигнал
        while unsafe { !core::ptr::read_volatile(&SIGNALLED) } {}
        // Сбрасываем индикатор сигнала
        unsafe { core::ptr::write_volatile(&mut SIGNALLED, false) };
        // Выполняем задачу, которая ждала прерывания
        run_task();
    }
}
```

В этом примере кода стоит отметить несколько моментов:

- Мы можем передать `&mut SIGNALLED` в функцию, требующую `*mut T`, поскольку `&mut T` автоматически преобразуется в `*mut T` (то же самое верно и для `*const T`);
- Нам нужны блоки `unsafe` для методов `read_volatile`/`write_volatile`, так как это `unsafe`-функции. Программист несёт ответственность за безопасное использование: подробности см. в документации к этим методам.

Редко приходится вызывать эти функции напрямую в своём коде, так как обычно о них позаботятся библиотеки более высокого уровня. Для регистров, отображённых в память, крейты доступа к периферии реализуют volatile-доступ автоматически, а для примитивов конкурентности есть более подходящие абстракции (см. главу о конкурентности).

10.0.6. Упакованные и выровненные типы

Во встраиваемом C часто требуется сообщить компилятору, что переменная должна иметь определённое выравнивание или что структура должна быть упакованной (packed), а не выровненной, обычно чтобы соответствовать требованиям конкретной аппаратуры или протокола.

В Rust это управляется атрибутом `repr` для структуры или объединения (union). Представление по умолчанию не даёт никаких гарантий о раскладке, поэтому его не следует использовать для кода, который взаимодействует с аппаратурой или C. Компилятор может переупорядочивать поля структуры или вставлять дополнительные байты заполнения (padding), и поведение может измениться в будущих версиях Rust.

```rust
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7ffecb3511d0 0x7ffecb3511d4 0x7ffecb3511d2
// 0x7ffecb3511d0 0x7ffecb3511d4 0x7ffecb3511d2
// Обратите внимание, что порядок изменён на x, z, y для лучшей упаковки.
```

Чтобы гарантировать раскладку, совместимую с C, используйте `repr(C)`:

```rust
#[repr(C)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
}

// 0x7fffd0d84c60 0x7fffd0d84c62 0x7fffd0d84c64
// 0x7fffd0d84c60 0x7fffd0d84c62 0x7fffd0d84c64
// Порядок сохранён, и раскладка не будет меняться со временем.
// `z` выровнен по двум байтам, поэтому между `y` и `z` есть один байт заполнения.
```

Чтобы гарантировать упакованное представление, используйте `repr(packed)`:

```rust
#[repr(packed)]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    // Ссылки всегда должны быть выровненными, поэтому чтобы проверить адреса
```

```
    // полей структуры, мы используем `std::ptr::addr_of!()`, чтобы получить сырой
    указатель,
    // вместо того чтобы просто выводить `&v.x`.
    let px = std::ptr::addr_of!(v.x);
    let py = std::ptr::addr_of!(v.y);
    let pz = std::ptr::addr_of!(v.z);
    println!("{:p} {:p} {:p}", px, py, pz);
}

// 0x7ffd33598490 0x7ffd33598492 0x7ffd33598493
// 0x7ffd33598490 0x7ffd33598492 0x7ffd33598493
// Между `y` и `z` не вставлено заполнение, поэтому теперь `z` не выровнено.
```

Обратите внимание, что использование `repr(packed)` также устанавливает выравнивание типа равным 1.

Наконец, чтобы задать конкретное выравнивание, используйте `repr(align(n))`, где `n` — количество байтов, по которым нужно выравнивать (и оно должно быть степенью двойки):

```Rust
#[repr(C)]
#[repr(align(4096))]
struct Foo {
    x: u16,
    y: u8,
    z: u16,
}

fn main() {
    let v = Foo { x: 0, y: 0, z: 0 };
    let u = Foo { x: 0, y: 0, z: 0 };
    println!("{:p} {:p} {:p}", &v.x, &v.y, &v.z);
    println!("{:p} {:p} {:p}", &u.x, &u.y, &u.z);
}

// 0x7ffec909a000 0x7ffec909a002 0x7ffec909a004
// 0x7ffec909b000 0x7ffec909b002 0x7ffec909b004
// 0x7ffec909a000 0x7ffec909a002 0x7ffec909a004
// 0x7ffec909b000 0x7ffec909b002 0x7ffec909b004
// Два экземпляра `u` и `v` размещены на границах 4096 байт,
// о чём свидетельствуют `000` в конце их адресов.
```

Обратите внимание, что `repr(C)` можно комбинировать с `repr(align(n))`, чтобы получить выровненную раскладку, совместимую с C. Нельзя комбинировать `repr(align(n))` с `repr(packed)`, так как `repr(packed)` устанавливает выравнивание равным 1. Также `repr(packed)`-тип не может содержать тип с `repr(align(n))`.

Подробнее о раскладке типов см. главу type layout справочника Rust.

10.0.7. Другие ресурсы

- В этой книге:
  - Немного C в вашем Rust
  - Немного Rust в вашем C
- Часто задаваемые вопросы о Rust для встраиваемых систем
- Указатели Rust для программистов на C
- I used to use pointers - now what?

11. Взаимодействие

Взаимодействие между кодом на Rust и C всегда зависит от преобразования данных между этими двумя языками. Для этой цели в `stdlib` есть специальный модуль `std::ffi`.

`std::ffi` предоставляет определения типов для примитивных типов C, таких как `char`, `int` и `long`. Он также содержит утилиты для преобразования более сложных типов, например строк, отображая `&str` и `String` на типы C, с которыми проще и безопаснее работать.

Начиная с Rust 1.30 функциональность `std::ffi` доступна либо в `core::ffi`, либо в `alloc::ffi`, в зависимости от того, связано ли это с выделением памяти. Крейт `cty` и крейт `cstr_core` также предлагают похожие возможности.

Тип Rust	Промежуточный тип	Тип C
`String`	`CString`	`char *`
`&str`	`CStr`	`const char *`
`()`	`c_void`	`void`
u32 или u64		

, `c_uint`, `unsigned int`, tr((en: [etc], de: [usw.], ja: [他], ru: [и т. д.],)), [...], [...],))

Значение примитивного типа C можно использовать как соответствующий тип Rust и наоборот, поскольку первый является просто псевдонимом типа второго. Например, следующий код скомпилируется на платформах, где `unsigned int` имеет 32 бита.

```Rust
fn foo(num: u32) {
    let c_num: c_uint = num;
    let r_num: u32 = c_num;
}
```

11.0.1. Взаимодействие с другими системами сборки

Распространённая задача при включении Rust в ваш встраиваемый проект — объединить Cargo с существующей системой сборки, например make или cmake.

Примеры и варианты использования для этого мы собираем в нашем трекере задач в issue #61.

11.0.2. Взаимодействие с RTOS

Интеграция Rust с RTOS, такой как FreeRTOS или ChibiOS, всё ещё находится в разработке; в особенности вызов функций RTOS из Rust может быть непростым.

На данный момент следующие проекты публично поддерживают взаимодействие Rust и RTOS:

- Zephyr Project

Примеры и варианты использования для этого мы собираем в нашем трекере задач в issue #62.

11.1. Немного C в вашем Rust

Использование C или C++ внутри проекта на Rust состоит из двух основных частей:

- оборачивание открытого C API для использования из Rust;
- сборка вашего C- или C++-кода, чтобы его можно было интегрировать с кодом на Rust.

Поскольку у C++ нет стабильного ABI, на который мог бы ориентироваться компилятор Rust, при объединении Rust с C или C++ рекомендуется использовать ABI `C`.

11.1.1. Определение интерфейса

Прежде чем использовать код на C или C++ из Rust, нужно определить (на Rust), какие типы данных и сигнатуры функций существуют в подключаемом коде. В C или C++ вы включили бы заголовочный файл (`.h` или `.hpp`), в котором определены эти данные. В Rust нужно либо вручную перевести эти определения на Rust, либо использовать инструмент, который их сгенерирует.

Сначала мы рассмотрим ручной перевод этих определений из C/C++ в Rust.

11.1.1.1. Оборачивание функций и типов данных C

Обычно библиотеки на C или C++ предоставляют заголовочный файл, в котором определены все типы и функции, используемые в открытых интерфейсах. Пример такого файла может выглядеть так:

```c
/* File: cool.h */
typedef struct CoolStruct {
    int x;
    int y;
} CoolStruct;

void cool_function(int i, char c, CoolStruct* cs);
```

При переводе на Rust этот интерфейс будет выглядеть так:

```rust
/* Файл: cool_bindings.rs */
#[repr(C)]
pub struct CoolStruct {
    pub x: core::ffi::c_int,
    pub y: core::ffi::c_int,
}

extern "C" {
    pub fn cool_function(
        i: core::ffi::c_int,
        c: core::ffi::c_char,
        cs: *mut CoolStruct
    );
}
```

Разберём это определение по частям, чтобы объяснить каждую из них.

```rust
#[repr(C)]
pub struct CoolStruct { ... }
```

По умолчанию Rust не гарантирует порядок, выравнивание (padding) или размер данных, входящих в `struct`. Чтобы гарантировать совместимость с C-кодом, мы добавляем атрибут `#[repr(C)]`, который указывает компилятору Rust всегда использовать для организации данных внутри структуры те же правила, что и C.

```
pub x: core::ffi::c_int,                                                    Rust
pub y: core::ffi::c_int,
```

Из-за гибкости в том, как C или C++ определяют `int` или `char`, рекомендуется использовать примитивные типы данных, определённые в `cty`: они сопоставляют типы C с типами Rust.

```
extern "C" { pub fn cool_function( ... ); }                                 Rust
```

Эта инструкция определяет сигнатуру функции, которая использует ABI C и называется `cool_function`. Поскольку определяется только сигнатура без тела функции, сама функция должна быть предоставлена где-то ещё или связана с итоговой библиотекой или бинарным файлом из статической библиотеки.

```
    i: core::ffi::c_int,                                                    Rust
    c: core::ffi::c_char,
    cs: *mut CoolStruct
```

Подобно нашему типу данных выше, мы определяем типы аргументов функции, используя совместимые с C определения. Для ясности мы сохраняем и те же имена аргументов.

Здесь есть один новый тип: `*mut CoolStruct`. Поскольку в C нет понятия ссылок Rust, которые выглядели бы так: `&mut CoolStruct`, мы вместо этого используем сырой указатель. Поскольку разыменование этого указателя — `unsafe`, а сам указатель может оказаться нулевым (`null`), нужно соблюдать осторожность, чтобы сохранить гарантии, типичные для Rust, при взаимодействии с кодом на C или C++.

11.1.1.2. Автоматическая генерация интерфейса

Вместо того чтобы вручную создавать эти интерфейсы, что может быть утомительно и чревато ошибками, существует инструмент bindgen, который выполнит эти преобразования автоматически. Инструкции по использованию bindgen см. в руководстве пользователя bindgen; обычно процесс состоит из следующих шагов:

1. Соберите все заголовочные файлы C или C++, которые определяют интерфейсы или типы данных, которые вы хотите использовать из Rust.
2. Напишите файл `bindings.h`, который `#include "..."` каждый из файлов, собранных на первом шаге.
3. Передайте этот файл `bindings.h` вместе с любыми флагами компиляции, которые использовались для сборки вашего кода, в `bindgen`. Совет: используйте `Builder.ctypes_prefix("cty")` / `--ctypes-prefix=cty` и `Builder.use_core()` / `--use-core`, чтобы сгенерированный код был совместим с `#![no_std]`.
4. `bindgen` выведет сгенерированный код Rust в терминал. Этот вывод можно перенаправить в файл в вашем проекте, например `bindings.rs`. Вы можете использовать этот файл в своём проекте на Rust для взаимодействия с кодом C/C++, скомпилированным и связанным как внешняя библиотека. Совет: не забудьте использовать крейт `cty`, если типы в сгенерированных привязках имеют префикс `cty`.

11.1.2. Сборка вашего кода на C/C++

Поскольку компилятор Rust не умеет напрямую компилировать код на C или C++ (как и код на любом другом языке, предоставляющем интерфейс C), код не на Rust нужно скомпилировать заранее.

Для встраиваемых проектов это чаще всего означает компиляцию кода на C/C++ в статический архив (например, `cool-library.a`), который затем объединяется с кодом на Rust на финальном шаге компоновки.

Если библиотека, которую вы хотите использовать, уже распространяется как статический архив, перекомпилировать код не нужно. Достаточно перевести заголовочный файл интерфейса, как описано выше, и подключить статический архив на этапе компиляции или компоновки.

Если ваш код существует как исходный проект, нужно скомпилировать код на C/C++ в статическую библиотеку: либо запустив вашу существующую систему сборки (например, `make`, `CMake` и т. д.), либо перенеся необходимые шаги компиляции на инструмент, называемый крейтом `cc`. Для обоих этих способов нужен скрипт `build.rs`.

11.1.2.1. Скрипты сборки `build.rs` на Rust

`build.rs` — это файл, написанный на синтаксисе Rust, который выполняется на вашей машине сборки ПОСЛЕ того, как собраны зависимости вашего проекта, но ДО сборки самого проекта.

Полный справочник находится здесь. Скрипты `build.rs` полезны для генерации кода (например, через bindgen), вызова внешних систем сборки, таких как `Make`, или прямой компиляции C/C++ с помощью крейта `cc`.

11.1.2.2. Запуск внешних систем сборки

Для проектов со сложными внешними проектами или системами сборки проще всего использовать `std::process::Command`, чтобы «вызвать» другие системы сборки, перейдя по относительным путям, запустив фиксированную команду (например, `make library`), а затем скопировав полученную статическую библиотеку в нужное место в каталоге сборки `target`.

Хотя ваш крейт может быть предназначен для встраиваемой платформы `no_std`, ваш `build.rs` выполняется только на машинах, которые собирают ваш крейт. Это означает, что вы можете использовать любые крейты Rust, которые работают на вашей машине сборки.

11.1.2.3. Сборка C/C++ кода с крейтом `cc`

Для проектов с ограниченными зависимостями или сложностью, а также для проектов, в которых трудно изменить систему сборки, чтобы создать статическую библиотеку (а не итоговый бинарный файл или исполняемый файл), может быть проще воспользоваться крейтом `cc`, который предоставляет идиоматичный интерфейс Rust к компилятору, предоставляемому хост-системой.

В простейшем случае, когда одиночный C-файл компилируется как зависимость в статическую библиотеку, пример скрипта `build.rs` с использованием крейта `cc` выглядит так:

```rust
fn main() {
    cc::Build::new()
        .file("src/foo.c")
        .compile("foo");
}
```

`build.rs` размещается в корне пакета. Затем `cargo build` скомпилирует и выполнит его перед сборкой пакета. Статический архив с именем `libfoo.a` будет создан и помещён в каталог `target`.

11.2. Немного Rust в вашем C

Использование кода на Rust внутри проекта на C или C++ в основном состоит из двух частей.

- Создание дружественного к C API на Rust.
- Встраивание вашего проекта на Rust во внешнюю систему сборки.

Кроме `cargo` и `meson`, большинство систем сборки не имеют встроенной поддержки Rust. Поэтому, скорее всего, лучше всего использовать `cargo` для компиляции вашего крейта и всех его зависимостей.

11.2.1. Настройка проекта

Создайте новый проект `cargo`, как обычно.

Есть флаги, которые указывают `cargo` выпускать системную библиотеку вместо обычной библиотеки Rust. Это также позволяет задать для вашей библиотеки имя вывода, отличное от остальной части крейта, если вы этого хотите.

```toml
[lib]
name = "your_crate"
crate-type = ["cdylib"]      # Создаёт динамическую библиотеку
# crate-type = ["staticlib"] # Создаёт статическую библиотеку
```

11.2.2. Построение API на C

Поскольку у C++ нет стабильного ABI, на который мог бы ориентироваться компилятор Rust, для любого взаимодействия между разными языками мы используем `C`. Это не исключение и при использовании Rust внутри кода на C и C++.

11.2.2.1. `#[no_mangle]`

Компилятор Rust искажает (mangle) имена символов иначе, чем ожидают нативные компоновщики. Поэтому любой функции, которую Rust экспортирует для использования за пределами Rust, нужно сообщить, что её не следует искажать.

11.2.2.2. `extern "C"`

По умолчанию любая функция, которую вы пишете на Rust, будет использовать ABI Rust (который тоже не стабилизирован). Вместо этого при создании внешних API FFI нужно сообщить компилятору использовать системный ABI.

В зависимости от вашей платформы вам может понадобиться указать конкретную версию ABI; они описаны здесь.

Если сложить эти части вместе, получится функция примерно такого вида:

```rust
#[no_mangle]
pub extern "C" fn rust_function() {

}
```

Точно так же, как при использовании кода `C` в вашем проекте на Rust, вам нужно преобразовывать данные в форму, которую будет понимать остальная часть приложения, и обратно.

11.2.3. Компоновка и общий контекст проекта

Итак, одна половина задачи решена. Как теперь это использовать?

Это очень сильно зависит от вашего проекта и/или системы сборки.

`cargo` создаст файл `my_lib.so`/`my_lib.dll` или `my_lib.a` в зависимости от вашей платформы и настроек. Эту библиотеку можно просто подключить с помощью вашей системы сборки.

Однако для вызова функции Rust из C нужен заголовочный файл, который объявляет сигнатуры функций.

Каждой функции вашего FFI-API на Rust нужно сопоставить соответствующее объявление в заголовке.

```rust
#[no_mangle]
pub extern "C" fn rust_function() {}
```

станет

```c
void rust_function();
```

и так далее.

Есть инструмент для автоматизации этого процесса — cbindgen, который анализирует ваш код на Rust и на его основе генерирует заголовки для ваших проектов на C и C++.

Теперь вызов функций Rust из C так же прост, как подключение заголовка и вызов функций!

```c
#include "my-rust-project.h"
rust_function();
```

12. Разные темы

12.1. Оптимизации: компромисс между скоростью и размером

Все хотят, чтобы их программа была одновременно очень быстрой и очень маленькой, но обычно невозможно получить обе эти характеристики. В этом разделе рассказывается о разных уровнях оптимизации, которые предоставляет `rustc`, и о том, как они влияют на время выполнения и размер бинарного файла программы.

12.1.1. Без оптимизаций

Это вариант по умолчанию. Когда вы вызываете `cargo build`, вы используете профиль разработки (dev). Этот профиль оптимизирован для отладки: он включает отладочную информацию и *не* включает никаких оптимизаций, то есть использует `-C opt-level = 0`.

По крайней мере для bare-metal разработки отладочная информация не имеет «стоимости» в том смысле, что она не занимает места во флеш-памяти / ПЗУ, поэтому мы фактически рекомендуем включить отладочную информацию в профиле release — по умолчанию она отключена. Это позволит вам использовать точки останова при отладке сборок release.

```toml
[profile.release]
# символы полезны, и они не увеличивают размер во флеш-памяти
debug = true
```

Отсутствие оптимизаций хорошо для отладки, потому что пошаговое выполнение кода ощущается так, будто вы выполняете программу оператор за оператором, и к тому же вы можете в GDB вывести значения стековых переменных и аргументов функций. Когда код оптимизирован, попытка вывести переменные приводит к тому, что печатается `$0 = <value optimized out>`.

Самый большой недостаток профиля `dev` в том, что итоговый бинарный файл будет огромным и медленным. Размер обычно создаёт больше проблем, потому что неоптимизированные бинарные файлы могут занимать десятки КиБ флеш-памяти, которой на вашем целевом устройстве может не хватить: в результате неоптимизированный бинарный файл просто не помещается в устройство!

Можно ли получить меньшие бинарные файлы, удобные для отладки? Да, есть один приём.

12.1.1.1. Оптимизация зависимостей

У Cargo есть возможность `profile-overrides`, которая позволяет переопределять уровень оптимизации зависимостей. С её помощью можно оптимизировать все зависимости по размеру, оставляя при этом верхний крейт неоптимизированным и удобным для отладки.

Имейте в виду, что обобщённый (generic) код иногда может оптимизироваться вместе с крейтом, в котором он создаётся, а не с крейтом, в котором он определён. Если вы создаёте экземпляр обобщённой структуры в своём приложении и обнаруживаете, что она подтягивает код большого объёма, возможно, увеличение уровня оптимизации соответствующих зависимостей не даст эффекта.

Вот пример:

```toml
# Cargo.toml
[package]
name = "app"
# ..
```

```
[profile.dev.package."*"] # +
opt-level = "z" # +
```

Без переопределения:

```
$ cargo size --bin app -- -A                                    Plain Text
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 9060   0x8000400
.rodata               1708   0x8002780
.data                    0  0x20000000
.bss                     4  0x20000000
```

С переопределением:

```
$ cargo size --bin app -- -A                                    Plain Text
app  :
section               size        addr
.vector_table         1024   0x8000000
.text                 3490   0x8000400
.rodata               1100   0x80011c0
.data                    0  0x20000000
.bss                     4  0x20000000
```

Это сокращение использования флеш-памяти примерно на 6 КиБ, причём верхний крейт остаётся удобным для отладки. Если вы зайдёте внутрь зависимости, то снова увидите сообщения `<value optimized out>`, но обычно вам нужно отлаживать верхний крейт, а не зависимости. А если вам *действительно* нужно отладить зависимость, вы можете воспользоваться `profile-overrides`, чтобы исключить конкретную зависимость из оптимизации. Пример ниже:

```
# ..                                                            TOML

# не оптимизируем крейт `cortex-m-rt`
[profile.dev.package.cortex-m-rt] # +
opt-level = 0 # +

# но оптимизируем все остальные зависимости
[profile.dev.package."*"]
codegen-units = 1 # лучшие оптимизации
opt-level = "z"
```

Теперь верхний крейт и `cortex-m-rt` удобны для отладки!

12.1.2. Оптимизация по скорости

По состоянию на 2018-09-18 `rustc` поддерживает три уровня оптимизации по скорости: `opt-level = 1`, 2 и 3. Когда вы запускаете `cargo build --release`, вы используете профиль release, который по умолчанию использует `opt-level = 3`.

И `opt-level = 2`, и `3` оптимизируют по скорости за счёт размера бинарного файла, но уровень `3` выполняет больше векторизации и инлайнинга, чем уровень `2`. В частности, вы увидите, что при `opt-level`, равном 2 или больше, LLVM разворачивает циклы. Разворачивание циклов имеет довольно высокую цену во флеш-памяти / ПЗУ (например, для цикла обнуления массива оно увеличивает код с 26 до 194 байтов), но при подходящих условиях (например, когда число итераций достаточно велико) может вдвое сократить время выполнения.

На данный момент нет способа отключить разворачивание циклов при `opt-level = 2` и `3`, поэтому, если вы не можете позволить себе его стоимость, оптимизируйте программу по размеру.

12.1.3. Оптимизация по размеру

По состоянию на 2018-09-18 `rustc` поддерживает два уровня оптимизации по размеру: `opt-level = "s"` и `"z"`. Эти названия унаследованы от clang / LLVM и не очень описательны, но `"z"` задумано так, чтобы создавать ещё меньшие бинарные файлы, чем `"s"`.

Если вы хотите, чтобы ваши бинарные файлы release были оптимизированы по размеру, измените настройку `profile.release.opt-level` в `Cargo.toml`, как показано ниже.

```toml
[profile.release]
# или "z"
opt-level = "s"
```

Эти два уровня оптимизации значительно снижают порог инлайнинга LLVM — метрику, которая используется, чтобы решить, инлайнить функцию или нет. Один из принципов Rust — абстракции с нулевой стоимостью; такие абстракции часто используют множество newtype-типов и маленьких функций для поддержания инвариантов (например, функций, которые заимствуют внутреннее значение, как `deref`, `as_ref`), поэтому низкий порог инлайнинга может помешать LLVM заметить возможности оптимизации (например, устранить мёртвые ветки или встроить вызовы замыканий).

При оптимизации по размеру стоит попробовать увеличить порог инлайнинга и посмотреть, влияет ли это на размер бинарного файла. Рекомендуемый способ изменить порог инлайнинга — добавить флаг `-C inline-threshold` к остальным rustflags в `.cargo/config.toml`.

```toml
# .cargo/config.toml
# предполагается, что вы используете шаблон cortex-m-quickstart
[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = [
  # ..
  "-C", "inline-threshold=123", # +
]
```

Какое значение использовать? По состоянию на 1.29.0 таковы пороги инлайнинга, которые используют разные уровни оптимизации:

- `opt-level = 3` использует 275
- `opt-level = 2` использует 225
- `opt-level = "s"` использует 75
- `opt-level = "z"` использует 25

При оптимизации по размеру стоит попробовать значения 225 и 275.

12.2. Математические вычисления в `#[no_std]`

Если вы хотите выполнять математические вычисления, например извлекать квадратный корень или экспоненту числа, и у вас доступна полная стандартная библиотека, ваш код может выглядеть так:

```rust
//! Некоторые математические функции с поддержкой стандартной библиотеки

fn main() {
    let float: f32 = 4.82832;
    let floored_float = float.floor();

    let sqrt_of_four = floored_float.sqrt();

    let sinus_of_four = floored_float.sin();

    let exponential_of_four = floored_float.exp();
    println!("Floored test float {} to {}", float, floored_float);
    println!("The square root of {} is {}", floored_float, sqrt_of_four);
    println!("The sinus of four is {}", sinus_of_four);
    println!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
}
```

Без поддержки стандартной библиотеки эти функции недоступны. Вместо них можно использовать внешний крейт, например `libm`. Тогда пример кода будет выглядеть так:

```rust
#![no_main]
#![no_std]

use panic_halt as _;

use cortex_m_rt::entry;
use cortex_m_semihosting::{debug, hprintln};
use libm::{exp, floorf, sin, sqrtf};

#[entry]
fn main() -> ! {
    let float = 4.82832;
    let floored_float = floorf(float);

    let sqrt_of_four = sqrtf(floored_float);

    let sinus_of_four = sin(floored_float.into());

    let exponential_of_four = exp(floored_float.into());
```

```
    hprintln!("Floored test float {} to {}", float, floored_float).unwrap();
    hprintln!("The square root of {} is {}", floored_float, sqrt_of_four).unwrap();
    hprintln!("The sinus of four is {}", sinus_of_four).unwrap();
    hprintln!(
        "The exponential of four to the base e is {}",
        exponential_of_four
    )
    .unwrap();
    // выход из QEMU
    // ПРИМЕЧАНИЕ: не запускайте это на реальном оборудовании; это может повредить
    состояние OpenOCD
    // debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Если вам нужно выполнять более сложные операции, например обработку сигналов (DSP) или продвинутую линейную алгебру на вашем MCU, могут помочь следующие крейты:

- Привязки к библиотеке CMSIS DSP
- `constgebra`
- `micromath`
- `microfft`
- `nalgebra`

A. Приложение A: Глоссарий

Экосистема встраиваемых систем полна разных протоколов, аппаратных компонентов и специфичных для производителей вещей, которые используют собственные термины и сокращения. Этот глоссарий пытается перечислить их и даёт ссылки для лучшего понимания.

BSP

Board Support Crate (BSP) предоставляет высокоуровневый интерфейс, настроенный для конкретной платы. Обычно он зависит от крейта HAL. Более подробное описание есть на странице про регистры, отображённые в память, а для общего обзора см. это видео.

FPU

Floating-point Unit (блок вычислений с плавающей точкой). «Математический процессор», который выполняет только операции с числами с плавающей точкой.

HAL

Hardware Abstraction Layer crate (крейт слоя аппаратной абстракции) предоставляет удобный для разработчика интерфейс к возможностям и периферии микроконтроллера. Обычно он реализуется поверх Peripheral Access Crate (PAC). Также он может реализовывать трейты из крейта `embedded-hal`. Более подробное описание есть на странице про регистры, отображённые в память, а для общего обзора см. это видео.

I2C

Иногда обозначается как `I²C` или Inter-IC. Это протокол для обмена данными между микросхемами на одной плате. Подробнее см. здесь.

PAC

Peripheral Access Crate (крейт доступа к периферии) предоставляет доступ к периферии микроконтроллера. Это один из крейтов самого низкого уровня, обычно он генерируется непосредственно из предоставленного файла SVD, часто с помощью svd2rust. Слой аппаратной абстракции обычно зависит от этого крейта. Более подробное описание есть на странице про регистры, отображённые в память, а для общего обзора см. это видео.

SPI

Serial Peripheral Interface (последовательный периферийный интерфейс). Подробнее см. здесь.

SVD

System View Description — формат файла на основе XML, который описывает представление программиста об устройстве микроконтроллера. Подробнее о нём можно прочитать на сайте документации ARM CMSIS.

UART

Universal asynchronous receiver-transmitter (универсальный асинхронный приёмопередатчик). Подробнее см. здесь.

USART

Universal synchronous and asynchronous receiver-transmitter (универсальный синхронный и асинхронный приёмопередатчик). Подробнее см. здесь.