**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#__RefHeading___Toc2163_169777778)

[1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ И АНАЛИЗ СУЩЕСТВУЮЩИХ РЕШЕНИЙ 7](#__RefHeading___Toc2165_169777778)

[1.1 Обзор предметной области 7](#__RefHeading___Toc2167_169777778)

[1.1.1 Введение в предметную область 7](#__RefHeading___Toc2169_169777778)

[1.1.2 Традиционный подход к реализации HAL 8](#__RefHeading___Toc2169_169777778_Copy_1)

[1.1.3 Реализация HAL на языке Rust 10](#__RefHeading___Toc2169_169777778_Copy_2)

[1.2 Анализ существующих решений 12](#__RefHeading___Toc2171_169777778)

[1.2.1 Mikron Mik32 HAL 12](#__RefHeading___Toc2173_169777778)

[1.2.2 stm32f1xx-hal 15](#__RefHeading___Toc2177_169777778)

[1.2.3 esp-idf-hal 18](#__RefHeading___Toc2179_169777778)

[1.2.4 Embassy 21](#__RefHeading___Toc2181_169777778_Copy_1)

[1.3 Выводы по разделу 24](#__RefHeading___Toc2183_169777778)

[2 ПОСТАНОВКА ЗАДАЧИ 25](#__RefHeading___Toc2185_169777778)

[2.1 Назначение разрабатываемого ПО 25](#__RefHeading___Toc2187_169777778)

[2.2 Формирование требований к библиотеке 25](#__RefHeading___Toc2189_169777778)

[2.3 Выбор средств разработки 25](#__RefHeading___Toc2191_169777778)

[3 РАЗРАБОТКА 26](#__RefHeading___Toc2193_169777778)

[3.1 Проектирование 26](#__RefHeading___Toc2195_169777778)

[3.2 Разработка 26](#__RefHeading___Toc2197_169777778)

[3.2.1 Разработка модуля xxxx 26](#__RefHeading___Toc2199_169777778)

[4 ДЕМОНСТРАЦИЯ РАБОТЫ БИБЛИОТЕКИ 27](#__RefHeading___Toc2201_169777778)

[4.1 Пример работы модуля xxxx 27](#__RefHeading___Toc2203_169777778)

[ЗАКЛЮЧЕНИЕ 28](#__RefHeading___Toc2205_169777778)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 29](#__RefHeading___Toc2207_169777778)

[ПРИЛОЖЕНИЯ 30](#__RefHeading___Toc2209_169777778)

[ПРИЛОЖЕНИЕ А. xxxx 30](#__RefHeading___Toc2211_169777778)

**ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ**

Микроконтроллер — микросхема для программного управления электронными устройствами.

Встраиваемая система (встроенная система, англ. embedded system) — специализированная микропроцессорная система управления, контроля и мониторинга, концепция разработки которой заключается в том, что такая система будет работать, будучи встроенной непосредственно в устройство, которым она управляет.

Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы.

Висячий указатель или висячая ссылка (англ. Dangling pointer, wild pointer, dangling reference) — указатель, не указывающий на допустимый объект соответствующего типа.

Гонка данных (англ. data race) — это состояние в многопоточных программах, когда два или более потоков одновременно пытаются читать и записывать в одну и ту же область памяти без должной синхронизации.

**ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ**

HAL (Hardware Abstraction Layer, слой аппаратных абстракций) — слой абстрагирования, реализованный в программном обеспечении, находящийся между физическим уровнем аппаратного обеспечения и программным обеспечением, запускаемом на устройстве.

RTOS (Real-time operating system, операционная система реального времени) — тип специализированной операционной системы, основное назначение которой — предоставление необходимого и достаточного набора функций для проектирования, разработки и функционирования систем реального времени на конкретном аппаратном оборудовании.

SVD (System View Description) — формат описания системного представления, который используется для формального описания регистров микроконтроллеров.

FFI (Foreign Function Interface, интерфейс внешних функций) — это механизм, с помощью которого программа, написанная на одном языке программирования, может вызывать подпрограммы, написанные на другом языке.

ВВЕДЕНИЕ

Современные микроконтроллеры являются ключевыми компонентами встроенных систем, обеспечивая управление широким спектром устройств – от простых датчиков до сложных промышленных контроллеров и интернета вещей. В условиях стремительного развития embedded-технологий особую важность приобретает не только аппаратная часть микроконтроллеров, но и программные средства для их эффективного использования. Одним из перспективных представителей данного класса устройств является микроконтроллер Mik32 Amur, обладающий сбалансированными характеристиками производительности, энергопотребления и стоимости.

Mik32 Amur является первым и единственным в России микроконтроллером первого уровня. Данный статус позволяет ему иметь приоритет в государственных закупках и гарантированную поставку в условиях ограниченной доступности зарубежных аналогов. Также это означает поддержку отечественной экосистемы разработки, включая инструменты отладки и программирования, и соответствие микроконтроллера требованиям информационной безопасности для применения в системах критической инфраструктуры.

Эффективное использование возможностей микроконтроллера в значительной степени зависит от качества программного обеспечения, в частности – от уровня аппаратной абстракции. HAL представляет собой промежуточный слой между аппаратными ресурсами микроконтроллера и пользовательским приложением, предоставляя удобный и безопасный интерфейс для работы с периферией. Наличие качественного HAL существенно сокращает время разработки, минимизирует количество ошибок, связанных с прямым доступом к регистрам, и облегчает портирование кода между различными платформами.

В последние годы язык программирования Rust приобретает все большую популярность в области разработки встроенного программного обеспечения. По сравнению с традиционными языками, такими как C и C++, Rust предлагает уникальное сочетание высокой производительности, строгой безопасности работы с памятью и современных возможностей метапрограммирования. Система владения и заимствования (ownership и borrowing) исключает целый класс ошибок, связанных с висячими указателями, некорректным доступом к памяти и гонками данных, что особенно критично в embedded-системах с ограниченными ресурсами. Кроме того, Rust предоставляет развитые средства для работы с низкоуровневыми конструкциями, позволяя писать эффективный код, близкий к железу, но без ущерба для безопасности и надёжности.

Актуальность данной работы обусловлена несколькими ключевыми факторами. Во-первых, несмотря на растущую популярность языка Rust в embedded-разработке, для многих микроконтроллеров, включая Mik32 Amur, отсутствуют готовые и полнофункциональные решения HAL на этом языке. Во-вторых, существующие реализации HAL для Mik32 Amur, как правило, написаны на C и не в полной мере используют современные возможности языков программирования, такие как строгая типизация, безопасность памяти и выразительные абстракции. В-третьих, переход на Rust в embedded-сфере соответствует общемировому тренду на повышение надёжности и безопасности программного обеспечения, особенно в критически важных системах.

Разработка HAL для Mik32 Amur на языке Rust позволит не только заполнить существующий пробел в инструментарии для данного микроконтроллера, но и продемонстрировать преимущества Rust перед традиционными языками embedded-разработки. Создаваемая библиотека сможет стать основой для будущих проектов на Mik32 Amur, обеспечивая разработчиков удобным, безопасным и эффективным инструментом для работы с аппаратными ресурсами микроконтроллера. Кроме того, данный проект внесёт вклад в развитие экосистемы Rust для встроенных систем, расширяя область его применения и демонстрируя его возможности на практике.

Таким образом, данная дипломная работа направлена на создание современного, надёжного и удобного HAL для микроконтроллера Mik32 Amur с использованием языка программирования Rust, что соответствует актуальным тенденциям в области разработки встроенного программного обеспечения и открывает новые возможности для сообщества embedded-разработчиков.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ И АНАЛИЗ СУЩЕСТВУЮЩИХ РЕШЕНИЙ

## Обзор предметной области

### Введение в предметную область

Слой аппаратных абстракций служит критически важным связующим звеном между физическим оборудованием и прикладным программным обеспечением в embedded-системах. Современные реализации HAL эволюционировали от простых библиотек до сложных фреймворков, предлагающих комплексные решения для управления аппаратными ресурсами. Основная функция HAL заключается в предоставлении унифицированного интерфейса для работы с периферийными устройствами, скрывая специфику аппаратной реализации за абстрактными программными интерфейсами.

Современные HAL обеспечивают безопасный доступ к оборудованию, контролируя корректность операций и предотвращая конфликты ресурсов через механизмы эксклюзивного доступа. Важной особенностью является гарантия временных характеристик выполнения операций, что особенно критично для систем реального времени. Кроме того, современные HAL оптимизируют производительность и энергопотребление, минимизируя накладные расходы и эффективно используя возможности устройств.

Современные тенденции в разработке HAL включают:

- поддержку многозадачности и интеграцию с RTOS,

- динамическую реконфигурацию оборудования во время выполнения,

- межплатформенную совместимость в рамках архитектурных семейств,

- использование генерации кода на основе метаданных,

- внедрение систем автоматического тестирования и верификации.

### Традиционный подход к реализации HAL

Традиционные реализации HAL преимущественно разрабатываются на языках C и C++, что обусловлено историческими предпосылками и сложившейся практикой embedded-разработки.

Архитектура классических HAL обычно строится вокруг следующих ключевых компонентов:

1. Регистровые обёртки

Регистровые обёртки - набор макроопределений или inline-функций, предоставляющих доступ к аппаратным регистрам. В типичной реализации это может выглядеть как серия define-директив, отображающих физические адреса регистров на символьные имена с битовыми масками для отдельных полей. Такой подход, несмотря на кажущуюся простоту, создает значительные риски некорректного использования, так как не обеспечивает проверку допустимых значений на этапе компиляции.

2. Драйверы периферии

Драйверы периферии обычно реализуются в виде набора функций, оперирующих с низкоуровневыми структурами данных. Например, драйвер UART может включать функции инициализации, передачи и приёма данных, управления потоком. Проблема таких реализаций заключается в отсутствии строгой типизации - параметры конфигурации передаются как простые числовые значения, что не позволяет компилятору выявлять логические ошибки.

3. Система обработки ошибок

Система обработки ошибок в традиционных HAL чаще всего строится на основе кодов возврата или глобальных переменных состояния. Это приводит к нескольким существенным недостаткам: невозможности гарантировать проверку ошибок на этапе компиляции, сложностям в отслеживании цепочек вызовов и потенциальным конфликтам при многопоточной работе.

4. Управление ресурсами

Управление ресурсами реализуется через явное выделение и освобождение, что создаёт риски утечек ресурсов или их повторного использования. Типичным примером может служить необходимость ручного вызова функций деинициализации периферийных модулей.

Особенно ярко недостатки традиционных подходов проявляются в следующих аспектах:

- безопасность работы с памятью - отсутствие защиты от переполнений буферов, висячих указателей и других распространённых уязвимостей. Например, передача некорректного размера буфера в функцию DMA может привести к порче памяти, что будет обнаружено только во время выполнения,

- проблемы конкурентного доступа - большинство традиционных HAL не предоставляет надёжных механизмов для работы в многопоточной среде, перекладывая ответственность за синхронизацию на разработчика приложения,

- сложность верификации - отсутствие выразительной системы типов затрудняет статический анализ кода и формальную верификацию, что особенно критично для систем с высокими требованиями к надежности,

- ограниченная расширяемость - жёсткая связность компонентов и отсутствие чётких интерфейсов затрудняют модификацию и адаптацию кода для новых версий микроконтроллеров.

При этом традиционные HAL продолжают широко использоваться благодаря:

- зрелости инструментальной поддержки,

- накопленной базе знаний и примеров,

- совместимости с унаследованными проектами.

Однако для современных проектов, особенно в областях с повышенными требованиями к безопасности и надёжности, эти подходы демонстрируют существенные ограничения, что стимулирует поиск альтернативных решений.

### Реализация HAL на языке Rust

Язык программирования Rust предлагает принципиально новые возможности для реализации HAL, сочетая низкоуровневый контроль с высокоуровневыми абстракциями. Ключевое преимущество Rust - система владения и заимствования, которая обеспечивает статическую гарантию эксклюзивного доступа к аппаратным ресурсам и предотвращает гонки данных без использования блокировок. Абстракции Rust позволяют проводить статическую верификацию конфигураций оборудования через систему типов, используя безразмерные типы для представления состояний устройств.

Инструментарий Rust открывает возможности для автоматической генерации кода на основе SVD-описаний, что значительно упрощает создание HAL для новых микроконтроллеров. Параметры шаблонов позволяют настраивать драйверы на этапе компиляции, а атрибуты - декларативно конфигурировать оборудование. Эргономика Rust проявляется в использовании системы признаков для унификации интерфейсов, модели ошибок на основе типа Result для явной обработки сбоев, а также асинхронных абстракций для эффективного управления событиями.

Интеграция с системой сборки Cargo и поддержка кросс-компиляции делают Rust особенно удобным для embedded-разработки. Отсутствие необходимости в сборщике мусора, возможность работы без стандартной библиотеки (атрибут no\_std) и точный контроль над размещением данных в памяти позволяют создавать эффективные реализации HAL, не уступающие по производительности традиционным C-решениям. При этом Rust-реализации демонстрируют значительно меньший уровень ошибок, связанных с памятью и ресурсами, а также улучшенную сопровождаемость и повторное использование кода.

Для Mik32 Amur применение Rust позволит:

- создать типобезопасные интерфейсы,

- исключить целые классы ошибок,

- построить современную экосистему.

## Анализ существующих решений

### Mikron Mik32 HAL

Официальная библиотека аппаратной абстракции для микроконтроллера Mik32 Amur производства компании "Миландр" представляет собой классический пример традиционного подхода к разработке HAL для встраиваемых систем. Реализация выполнена на языке C и следует архитектурным принципам, характерным для большинства HAL от производителей микроконтроллеров.

Библиотека разделена на 2 основных модуля:

1. mik32v2-shared

mik32v2-shared содержит следующие элементы:

- описание карты памяти микроконтроллера,

- описания периферийных регистров,

- низкоуровневые интерфейсы для работы с периферией,

- скрипты для компоновщика,

- ассемблерный код для инициализации микроконтроллера после запуска.

2. mik32-hal

mik32-hal содержит следующие элементы:

- библиотеку системного таймера ядра,

- библиотеки для программирования периферийных блоков Mik32,

- библиотеки поддержки сторонних устройств.

Кодовая база использует следующие подходы:

- регистровые макросы - доступ к периферии через набор define-директив:

|  |
| --- |
| #define WDT (( WDT \*) WDT\_BASE\_ADDRESS)  #define UART\_0 (( UART\_\*) UART\_0\_BASE\_ADDRESS) |

- функциональные API — набор функций для работы с периферией:

|  |
| --- |
| void HAL\_USART\_MspInit(USART\_HandleTypeDef\* setting);  HAL\_StatusTypeDef HAL\_USART\_Init(USART\_HandleTypeDef\* setting); |

- структуры конфигурации: передача параметров инициализации

|  |
| --- |
| typedef struct \_\_GPIO\_InitTypeDef  {  HAL\_PinsTypeDef Pin;  HAL\_GPIO\_PullTypeDef Pull;  HAL\_GPIO\_DSTypeDef DS;  } GPIO\_InitTypeDef; |

Ключевые преимущества официальной реализации HAL для Mik32 Amur заключаются в её высокой эффективности и практичности для традиционной embedded-разработки. Решение обеспечивает прямой низкоуровневый доступ к аппаратным ресурсам с минимальными накладными расходами, что особенно важно для задач с жёсткими требованиями к производительности. Библиотека демонстрирует отличную совместимость с распространёнными средами разработки, предлагая знакомый многим инженерам интерфейс, схожий с HAL для популярных микроконтроллеров STM32.

Основные ограничения текущей реализации связаны с устаревшей архитектурой и отсутствием современных механизмов обеспечения надёжности. Главной проблемой остаётся слабая защищенность от ошибок разработчика — отсутствие проверок границ, возможные конфликты ресурсов и примитивная система обработки ошибок существенно повышают риски возникновения критических сбоев. С точки зрения эргономики разработки, решение страдает от неудобной системы конфигурации и практически полного отсутствия типобезопасности, что значительно увеличивает вероятность ошибок на этапе компиляции.

Пример типичного использования:

|  |
| --- |
| #include "mik32\_hal\_gpio.h"  #include "mik32\_hal\_pcc.h"  int main() {  PCC\_InitTypeDef PCC\_OscInit = {0};  PCC\_OscInit.OscillatorEnable = PCC\_OSCILLATORTYPE\_ALL;  PCC\_OscInit.FreqMon.OscillatorSystem = PCC\_OSCILLATORTYPE\_OSC32M;  PCC\_OscInit.HSI32MCalibrationValue = 128;  PCC\_OscInit.LSI32KCalibrationValue = 8;  HAL\_PCC\_Config(&PCC\_OscInit);  GPIO\_InitTypeDef GPIO\_InitStruct = {0};  \_\_HAL\_PCC\_GPIO\_2\_CLK\_ENABLE();  GPIO\_InitStruct.Pin = GPIO\_PIN\_7;  GPIO\_InitStruct.Mode = HAL\_GPIO\_MODE\_GPIO\_OUTPUT;  HAL\_GPIO\_Init(GPIO\_2, &GPIO\_InitStruct);  while (1) {  HAL\_GPIO\_TogglePin(GPIO\_2, GPIO\_PIN\_7);  HAL\_DelayMs(500);  }  } |

### stm32f1xx-hal

Библиотека stm32f1xx-hal представляет собой эталонную реализацию аппаратной абстракции для семейства микроконтроллеров STM32F1, разработанную сообществом Rust-разработчиков для embedded-систем. Этот проект служит важным ориентиром при создании HAL для других архитектур, демонстрируя передовые практики embedded-разработки на языке Rust.

Библиотека построена на трёх фундаментальных концепциях, которые радикально отличают её от традиционных C-реализаций HAL. В основе архитектуры лежит глубокое переосмысление взаимодействия с аппаратными ресурсами через призму возможностей языка Rust:

1. Система типов

Система типов как основа безопасности превращает обычные операции с периферией в строго регламентированные процессы. Состояния устройств кодируются на уровне системы типов, где каждый переход между режимами работы сопровождается проверкой корректности на этапе компиляции. Например, попытка чтения из пина, сконфигурированного как выход, будет отвергнута компилятором ещё до передачи кода на микроконтроллер.

2. Принцип владения ресурсами

Принцип владения ресурсами пронизывает всю архитектуру библиотеки. Каждое периферийное устройство становится уникальным объектом, существующим в строго определённом состоянии. Этот подход исключает классические проблемы embedded-разработки - двойную инициализацию, конфликты доступа и некорректные последовательности управления. Ресурсы не просто выделяются и освобождаются - их жизненный цикл становится частью статически проверяемой логики программы.

3. Единая экосистема взаимодействия

Единая экосистема взаимодействия достигается через строгое следование embedded-hal стандартам. Библиотека не существует изолированно - она становится частью расширяемого мира Rust embedded компонентов. Интерфейсы спроектированы таким образом, что драйверы устройств могут работать с различными семействами микроконтроллеров без изменения кода. Это достигается через систему трейтов, где конкретная реализация скрыта за стандартизированным поведением.

Библиотека балансирует между низкоуровневым контролем и высокоуровневыми абстракциями. Разработчик получает доступ ко всем возможностям микроконтроллера, но при этом защищён от потенциально опасных операций. Такой подход требует значительных усилий при проектировании, но окупается существенным снижением количества ошибок в конечных продуктах.

Основные преимущества данной реализации заключаются в её принципиально новом подходе к проектированию HAL, который стал возможен благодаря уникальным особенностям языка Rust. Библиотека демонстрирует качественно новый уровень безопасности и надёжности за счёт глубокой интеграции механизмов владения и заимствования в работу с аппаратными ресурсами. Система типов Rust позволяет выражать состояния и конфигурации оборудования на уровне типов, что исключает целые классы ошибок ещё на этапе компиляции. При этом сохраняется высокая производительность, характерная для традиционных C-реализаций, благодаря нулевой стоимости абстракций и эффективной работе оптимизатора.

Значимым достижением является полная интеграция с экосистемой Rust embedded, включая поддержку стандартных embedded-hal интерфейсов, что обеспечивает совместимость с широким спектром устройств и драйверов. Развитая система документации с многочисленными примерами и активное сообщество разработчиков значительно снижают порог входа по сравнению с другими Rust-решениями для embedded.

Основные недостатки реализации связаны с процессом обучения — для эффективной работы с библиотекой требуется не только понимание embedded-разработки, но и глубокое знание системы типов Rust и особенностей его компилятора. Практическое использование сталкивается с определёнными сложностями при отладке ошибок компилятора, которые могут быть неочевидны для разработчиков, привыкших к традиционным языкам embedded-программирования.

Пример типичного использования:

|  |
| --- |
| #![no\_main]  #![no\_std]  use panic\_halt as \_;  use cortex\_m\_rt::entry;  use cortex\_m\_semihosting::hprintln;  use stm32f1xx\_hal::{pac, prelude::\*};  #[entry]  fn main() -> ! {  let p = pac::Peripherals::take().unwrap();  let mut crc = p.CRC.new();  crc.reset();  crc.write(0x12345678);  let val = crc.read();  hprintln!("found={:08x}, expected={:08x}", val, 0xdf8a8a2b\_u32);  loop {}  } |

### esp-idf-hal

esp-idf-hal – это слой аппаратных абстракций для микроконтроллеров семейства ESP32, написанный на языке Rust. Он предоставляет безопасные и идиоматичные Rust-интерфейсы для работы с периферией микроконтроллера, используя в качестве основы официальный фреймворк ESP-IDF (Espressif IoT Development Framework). esp-idf-hal разработана для интеграции Rust в экосистему ESP32, предоставляя альтернативу традиционному программированию на C/C++ с ESP-IDF.

Библиотека esp-idf-hal демонстрирует комплексный подход к интеграции Rust с аппаратной платформой ESP32. В её основе лежит оригинальная архитектура, которая умело сочетает безопасность современных языковых возможностей Rust с требованиями embedded-разработки.

Гибридная модель абстракции представляет собой тщательно продуманную систему уровней. На нижнем уровне реализовано взаимодействие с ESP-IDF через механизмы FFI, что позволяет сохранить полную совместимость с существующей экосистемой. Над этим уровнем построен промежуточный слой, который трансформирует "сырые" C-интерфейсы в безопасные Rust-обёртки. Венчает эту конструкцию верхний уровень абстракции, предлагающий разработчикам удобные идиоматичные API, соответствующие стандартам embedded-hal.

Асинхронная модель работы построена с учётом специфики embedded-систем. Библиотека предлагает разработчикам выбор между традиционным синхронным API и современным асинхронным подходом. Реализация включает специализированные оптимизации, такие как локальные исполнители задач для каждого периферийного модуля и глубокую интеграцию с механизмом пробуждения (Waker) ESP-IDF.

Поддержка периферии организована через единую систему доступа, основанную на паттерне Singleton. Это обеспечивает безопасное управление ресурсами микроконтроллера. Для разных семейств чипов ESP32 реализованы адаптеры, которые предоставляют общий интерфейс работы с сохранением специфических особенностей каждого процессора.

Система управления зависимостями отличается гибкостью и продуманностью. Библиотека использует механизмы условной компиляции, позволяющий включать только необходимые компоненты. Реализована модель ленивой инициализации, когда системные ресурсы выделяются только при первом обращении, что особенно важно для embedded-устройств с ограниченной памятью.

Особую роль играет подсистема обработки ошибок, которая выходит за рамки стандартных подходов. Создана иерархическая классификация ошибок, учитывающая их природу (аппаратные, логические, системные). Реализовано автоматическое контекстное логгирование, значительно упрощающее отладку сложных сценариев.

esp-idf-hal представляет собой удачный компромисс между безопасностью Rust и широкими возможностями экосистемы ESP-IDF. Библиотека предоставляет разработчикам идиоматичные Rust-интерфейсы для работы с периферией ESP32, сохраняя при этом доступ ко всем функциям оригинального SDK от Espressif. Такой подход позволяет создавать надёжные embedded-приложения, но при этом не ограничивая разработчика в выборе доступных функций микроконтроллера.

Одним из наиболее значимых достоинств является поддержка асинхронного программирования через интеграцию с esp-idf-svc. Это открывает возможности для создания эффективных event-driven приложений, работающих с сетевыми интерфейсами и различными периферийными устройствами. При этом разработчик получает доступ к стандартной библиотеке Rust, что существенно упрощает реализацию сложной логики по сравнению с традиционными no\_std подходами.

Несмотря на впечатляющие возможности, esp-idf-hal имеет ряд существенных ограничений. Тесная интеграция с ESP-IDF означает необходимость работы с C-компонентами, что усложняет процесс сборки и отладки. Разработчикам приходится иметь дело с дополнительными уровнями абстракции, что неизбежно сказывается на размере конечного бинарника и в некоторых случаях - на производительности.

Поддержка новых микроконтроллеров семейства ESP часто запаздывает, что может стать серьёзным ограничением для проектов, требующих использования актуальных аппаратных платформ. Кроме того, глубокая интеграция с ESP-IDF делает решения на основе esp-idf-hal менее переносимыми и более зависимыми от конкретной экосистемы.

Пример типичного использования:

|  |
| --- |
| use esp\_idf\_hal::delay::FreeRtos;  use esp\_idf\_hal::gpio::\*;  use esp\_idf\_hal::peripherals::Peripherals;  fn main() -> anyhow::Result<()> {  esp\_idf\_hal::sys::link\_patches();  let peripherals = Peripherals::take()?;  let mut led = PinDriver::output(peripherals.pins.gpio4)?;  loop {  led.set\_high()?;  // we are sleeping here to make sure the watchdog isn't triggered  FreeRtos::delay\_ms(1000);  led.set\_low()?;  FreeRtos::delay\_ms(1000);  }  } |

### Embassy

Библиотека Embassy представляет собой современный фреймворк для embedded-разработки на Rust, построенный вокруг асинхронной парадигмы. Его архитектура радикально отличается от традиционных подходов, предлагая принципиально новый способ работы с микроконтроллерами.

Ядро исполнительной системы составляет наиболее инновационную часть Embassy. В отличие от классических RTOS, здесь реализована легковесная модель выполнения, основанная на асинхронных задачах и кооперативной многозадачности. Особенностью является полный отказ от вытесняющей многозадачности в пользу более предсказуемой модели совместного выполнения. Это позволяет достичь исключительной эффективности даже на маломощных устройствах.

Интеграция с аппаратным обеспечением реализована через систему HAL-интерфейсов, которые обеспечивают унифицированный доступ к периферии. Embassy предлагает собственную реализацию embedded-hal с акцентом на асинхронные операции. Например, UART-интерфейс представлен не традиционными блокирующими функциями, а асинхронными методами с поддержкой await.

Система управления временем построена вокруг специализированных асинхронных таймеров. В отличие от обычных подходов с callback-функциями, Embassy использует механизм Future, что позволяет писать линейный код, который компилятор преобразует в конечный автомат.

Работа с прерываниями переосмыслена в рамках асинхронной парадигмы. Вместо обработчиков прерываний традиционного вида, Embassy предлагает концепцию "асинхронных прерываний", где ожидание события оформляется как обычный await. Это кардинально упрощает написание корректного конкурентного кода.

Поддержка различных архитектур реализована через систему адаптеров. На текущий момент Embassy поддерживает ARM Cortex-M, RISC-V и частично xtensa (ESP32). Для каждой архитектуры предоставляется оптимизированный исполнитель (executor), учитывающий особенности процессора.

Экосистема периферийных драйверов строится вокруг принципа "асинхронности по умолчанию". В отличие от других HAL, где асинхронность добавляется как опция, в Embassy все драйверы изначально проектируются для работы в асинхронном контексте. Это обеспечивает единообразие API и предсказуемое поведение.

Система управления питанием интегрирована непосредственно в исполнитель. Когда нет активных задач, система автоматически переводит процессор в режим пониженного энергопотребления, что критически важно для батарейных устройств. При этом пробуждение происходит автоматически при наступлении ожидаемых событий.

Особенности компиляции заслуживают отдельного внимания. Embassy активно использует возможности Rust для статического анализа и оптимизации. В результате получается крайне компактный код, где размер асинхронной задачи часто сопоставим с рукописным ассемблерным кодом.

Embassy предлагает принципиально новый подход к embedded-разработке, где асинхронная парадигма становится основой всего фреймворка. Главная сила библиотеки — в органичном сочетании высокой производительности и выразительности кода. Использование async/await позволяет писать линейный и понятный код, который при этом эффективно работает даже на маломощных устройствах.

Основным преимуществом Embassy является энергоэффективность — интегрированная система управления питанием автоматически оптимизирует энергопотребление, что критически важно для батарейных устройств. При этом фреймворк сохраняет предсказуемость реального времени, что выгодно отличает его от традиционных RTOS.

Главный компромисс Embassy — в требовании полного перехода на асинхронную парадигму. Это создает определенный порог входа для разработчиков, привыкших к традиционным синхронным подходам. Некоторые низкоуровневые операции могут требовать неочевидных решений при интеграции с legacy-кодом.

Другое существенное ограничение — текущая поддержка аппаратных платформ. Хотя основные архитектуры (Cortex-M, RISC-V) охвачены, поддержка некоторых специфичных чипов и периферии пока остается ограниченной. Это может потребовать дополнительных усилий при работе с экзотическим железом.

Пример типичного использования:

|  |
| --- |
| #![no\_std]  #![no\_main]  use defmt::info;  use embassy\_executor::Spawner;  use embassy\_stm32::Config;  use embassy\_time::Timer;  use {defmt\_rtt as \_, panic\_probe as \_};  #[embassy\_executor::main]  async fn main(\_spawner: Spawner) -> ! {  let config = Config::default();  let \_p = embassy\_stm32::init(config);  loop {  info!("Hello World!");  Timer::after\_secs(1).await;  }  } |

## Выводы по разделу

Проведённый анализ существующих HAL-библиотек для различных микроконтроллерных платформ позволил выявить ключевые архитектурные подходы, которые могут быть применены при разработке HAL для Mik32 Amur. Каждая из рассмотренных реализаций демонстрирует уникальные решения общих проблем embedded-разработки. В таблице 1.1 представлено краткое сравнение решений.

Таблица 1.1 – Краткое сравнение существующий решений

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Решение**  **Функция** | **Mikron Mik32 HAL** | **stm32f1xx-hal** | **esp-idf-hal** | **Embassy** |
| Парадигма | Синхронная | Синхронная | Смешанная | Асинхронная |
| Уровень абстракции | Низкий | Средний | Высокий | Высокий |
| Безопасность | Минимальная | Высокая | Высокая | Высокая |
| Поддержка RTOS | Отсутствует | RTIC | FreeRTOS | Встроенная, async/await |
| Кроссплатформенность | Только Mik32 | Только STM32F1 | Только ESP32 | Cortex-M, RISC-V, Xtensa |
| Поддержка | Ограниченная | Активное сообщество | Активное сообщество | Быстрорастущее сообщество |

Разработка HAL для Mik32 Amur требует сбалансированного подхода, объединяющего ключевые преимущества существующих решений.

1. ПОСТАНОВКА ЗАДАЧИ

## Назначение разрабатываемого ПО

## Формирование требований к библиотеке

## Выбор средств разработки

xxxx

1. РАЗРАБОТКА

## Проектирование

## Разработка

### Разработка модуля xxxx

xxxx

1. ДЕМОНСТРАЦИЯ РАБОТЫ БИБЛИОТЕКИ

## Пример работы модуля xxxx

ЗАКЛЮЧЕНИЕ

Xxxx

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Микроконтроллер // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Микроконтроллер> (дата обращения: 01.06.2025)
2. Встраеваемая система // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Встраиваемая_система> (дата обращения: 01.06.2025)
3. Слой аппаратных абстракций // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Слой_аппаратных_абстракций> (дата обращения: 01.06.2025)
4. Метапрограммирование // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Метапрограммирование> (дата обращения: 01.06.2025)
5. Висячий указатель // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Висячий_указатель> (дата обращения: 01.06.2025)
6. Data Race // [Электронный ресурс]: SecurityLab. – URL: <https://www.securitylab.ru/glossary/data_race/> (дата обращения: 01.06.2025)
7. Операционная система реального времени // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Операционная_система_реального_времени> (дата обращения: 01.06.2025)
8. System View Description // [Электронный ресурс]: ARM Software. – URL: <https://arm-software.github.io/CMSIS_5/SVD/html/index.html> (дата обращения: 01.06.2025)
9. Интерфейс внешних функций // [Электронный ресурс]: Wikipedia. – URL: <https://ru.wikipedia.org/wiki/Интерфейс_внешних_функций> (дата обращения: 01.06.2025)

ПРИЛОЖЕНИЯ

## ПРИЛОЖЕНИЕ А. xxxx