Embedded Technoloy and Intelligent Systems 嵌入式技术与智能系统, 2025, 2(3), 176-195
Published Online October 2025 in Hans. https://www.hanspub.org/journal/etis
https://doi.org/10.12677/etis.2025.23015

Hans汉斯

RusT-Thread：基于Rust面向资源受限嵌入式设备的操作系统的实践

罗浩民，陈琳波，刘　时，李　丁，赵于洋

中国科学技术大学计算机科学与技术学院，安徽　合肥

收稿日期：2025年9月22日；录用日期：2025年10月21日；发布日期：2025年10月30日

摘　要

随着物联网和嵌入式系统的发展，实时操作系统(RTOS)的安全性和性能需求日益提高。传统基于C语言的RTOS在内存安全和并发控制方面存在局限，容易导致缓冲区溢出、数据竞争等问题。本项目以RT-Thread为基础，使用Rust语言重构其内核，形成了全新的RusT-Thread系统。系统采用模块化架构，涵盖内核服务、进程调度、内存管理、线程通信与时钟控制等核心功能，并充分利用Rust的所有权模型与类型系统，实现内存安全与并发安全保障。项目创新性地引入改进的多级反馈队列调度算法、中断安全数据容器(RTIntrFreeCell)、内联汇编与动态－静态数据分离等技术，在保证功能兼容性的同时优化了代码简洁性与可维护性。通过单元测试、集成测试和性能基准测试，RusT-Thread在中断延时、上下文切换和线程创建时间等关键指标上表现出与RT-Thread相当甚至更优的实时性能。该工作不仅展示了Rust在系统软件开发中的可行性与优势，也为未来安全可靠的嵌入式RTOS设计提供了参考。

关键词

Rust，RTOS，RT-Thread，内存安全，并发安全，多级反馈队列调度，嵌入式系统

RusT-Thread: A Rust-Based Operating System for Resource-Constrained Embedded Devices

Haomin Luo, Linbo Chen, Shi Liu, Ding Li, Yuyang Zhao

School of Computer Science and Technology, University of Science and Technology of China, Hefei Anhui

Received: September 22, 2025; accepted: October 21, 2025; published: October 30, 2025

文章引用: 罗浩民, 陈琳波, 刘时, 李丁, 赵于洋. RusT-Thread: 基于 Rust 面向资源受限嵌入式设备的操作系统的实践[J]. 嵌入式技术与智能系统, 2025, 2(3): 176-195. DOI: 10.12677/etis.2025.23015

Abstract

With the rapid development of IoT and embedded systems, the requirements for the security and performance of real-time operating systems (RTOS) are increasing. Traditional C-based RTOS implementations suffer from limitations in memory safety and concurrency control, which can lead to buffer overflows, data races, and system instability. This project reconstructs the RT-Thread operating system entirely in Rust, resulting in a new RusT-Thread system. The system adopts a modular architecture covering kernel services, process scheduling, memory management, inter-thread communication, and clock control, while leveraging Rust's ownership model and type system to ensure memory and concurrency safety. Key innovations include an improved multi-level feedback queue scheduling algorithm, an interrupt-safe data container (RTIntrFreeCell), inline assembly integration, and a dynamic–static data separation design, which optimizes code simplicity and maintainability while ensuring functional compatibility. Comprehensive validation through unit tests, integration tests, and performance benchmarks demonstrates that RusT-Thread achieves real-time performance comparable to or even better than RT-Thread in terms of interrupt latency, context switching, and thread creation time. This work highlights the feasibility and advantages of Rust in system software development and provides valuable insights for the design of secure and reliable embedded RTOS in the future.

Keywords

Rust, RTOS, RT-Thread, Memory Safety, Concurrency Safety, Multi-Level Feedback Queue Scheduling, Embedded Systems

1. 简介

RusT-Thread 是一款基于 RT-Thread 理念，采用 Rust 语言打造的轻量级实时操作系统内核，已在 ARM Cortex-M4 上成功实现，并有望拓展至更多芯片平台。

在线程调度上，它支持多线程创建、调度及优先级管理，提供多种调度算法，如优先级、优先级 + RR 等。同时，在内存分配上支持伙伴系统、小内存分配器等多种内存分配方式，满足动态内存分配需求。此外，还实现了高精度定时器，支持单次和周期定时功能，并具备完整的异常和中断处理机制。

RusT-Thread 的 Rust 实现保障内存安全，模块化设计便于扩展和移植，支持资源受限的嵌入式系统，为开发者提供了安全、高效、精简的 Rust 原生实时操作系统选择。

本文将从项目特色、架构设计、核心实现机制、性能验证等多个维度，详细阐述 RusT-Thread 的设计理念与实现方式，展示其如何将 Rust 的现代化语言优势与实时操作系统的经典思想相结合，为嵌入式开发者提供一个更安全、更高效的开发新选择。

2. 项目特色

本项目是一个完全由 Rust 构建的操作系统内核。不同于其他使用 Rust 重构操作系统的工作，我们放弃了 C 和 Rust 混合编译改写操作系统内核的开发路径，而是选择从底层开始就用 Rust 开发，这种彻底的重构能让我们更好地利用 Rust 拥有的现代化语言特性，组织起结构更加清晰，功能实现更加简练的代

码。同时，我们还避免了混合编译过程中的各种操作性问题和潜在的安全性风险。

本项目的架构参考 RT-Thread nano 内核实现，在实现功能模块时我们保留了大部分 RT-Thread 内核的接口[1] [2]，确保熟悉 RT-Thread 内核的开发者能低成本快速上手我们的内核。RT-Thread 的成功很难离开其众多贡献者带来的丰富软件包移植，我们希望我们的内核也能为 RT-Thread 社区中的 Rust 开发者提供一个底层平台，可以原生地用 Rust 实现各种组件，丰富这个操作系统的功能。

3. 架构概述

当我们着手设计 RusT-Thread 架构时，我们的目标并不仅仅是复刻 RT-Thread，而是要用 Rust 语言的思想去重塑它，我们设计的总体架构如图 1 所示。为此，我们将以下几个关键原则融入了项目里：

- **安全性**：借助 Rust 的所有权系统与借用检查机制，从编译阶段就彻底消除内存安全隐患。同时，通过其严格的并发模型有效防止数据竞争，能够有效提升系统的并发安全性。
- **可扩展性**：我们将整个系统设计为高度模块化。这种架构使得无论是添加新的设备驱动、文件系统，还是集成复杂的网络协议栈，都变得直观而高效，为未来的功能拓展留出了充足的空间。
- **性能**：充分利用 Rust 的零成本抽象特性，在确保系统安全性的基础上，对调度算法和内存管理机制进行深度优化，从而全面提升系统的整体性能表现。
- **易移植性**：我们采用分层设计策略并构建清晰的硬件抽象层，简化了系统对不同芯片架构的适配过程。目前，该系统已成功支持 ARM Cortex-M4 芯片架构，未来也许会逐步扩展对更多类型芯片的支持范围。

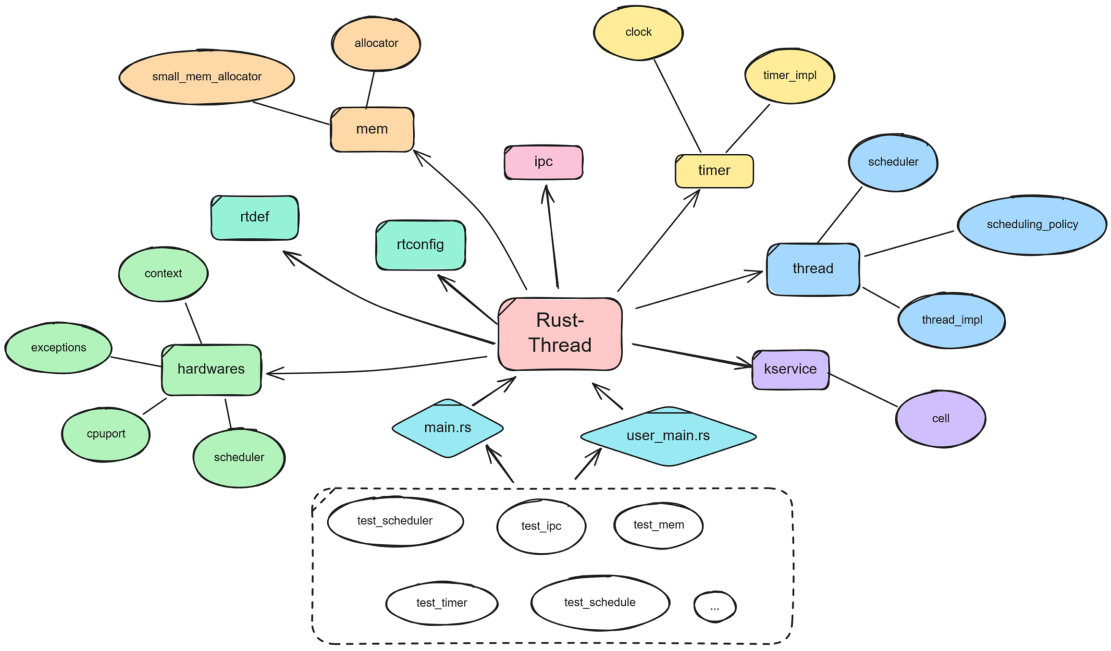

Figure 1. RusT-Thread overall architecture
图 1. RusT-Thread 总体架构图

4. RusT-Thread 模块介绍

4.1. 内核服务层

内核服务层作为 RusT-Thread 操作系统的核心基础，是连接硬件底层与上层应用的关键桥梁，其核

心功能及在 Rust 中的实现方案如下：

- **错误处理机制**：采用 Rust 的 Option 和 Result 枚举类型替代传统错误码，通过模式匹配清晰表达函数执行结果。在编译期对错误处理逻辑进行严格检查，避免因错误处理不当引发的问题，确保系统稳定运行。
- **内存操作函数**：利用 Rust 标准库中的 Core::alloc 模块，结合自定义的内存分配策略和数据结构，实现高效灵活的内存分配与回收机制，优化内存使用效率，确保操作的正确性和安全性。
- **字符串操作函数**：借助 Rust 内置的 String 和 str 类型及其操作方法，实现功能强大且安全高效的字符串处理功能，有效防止缓冲区溢出等问题，满足系统对文本处理的需求。
- **格式化输出功能**：结合 cortex_m_semihosting 工具库和 Rust 的格式化字符串宏(format!、println!等)，实现数据格式化输出功能。在 QEMU 模拟器环境下，通过半宿主功能将输出数据发送至宿主机控制台，支持多种数据类型和输出格式，满足调试和信息展示的需求。
- **调试和断言功能**：借助 Rust 的 Debug 和 Display 特性以及 assert!、debug_assert!等宏，实现完善的调试和断言功能。为自定义数据类型实现相关特质，以便在调试输出时展示详细信息；通过断言条件检查及时暴露问题，提高开发效率。
- **链表实现**：利用 Rust 的 Vec 等标准容器类型，结合手动指针操作和数据结构管理逻辑，构建高效的链表结构。通过存储节点指针模拟链接关系，实现节点的动态操作，同时借助所有权系统和借用检查机制确保操作的安全性和正确性。

我们单独添加了 Cell 模块，以更好地适配 Rust 的语言特性。Cell 模块实现了一个中断安全的共享数据容器 RTIntrFreeCell<T>，这是 RT-Thread 实时操作系统中的核心内核服务组件之一。其核心目的就是在多线程和中断环境下提供安全的共享数据访问机制，通过自动禁用和启用中断来防止数据竞争，具体体现在以下三个方面：

1. 中断安全性

- 在访问共享数据时自动调用 rt_hw_interrupt_disable ()禁用中断
- 访问结束后自动调用 rt_hw_interrupt_enable ()恢复中断
- 确保临界区代码能够原子性地执行，避免数据不一致的问题

2. RAII 资源管理

- 使用 RTIntrRefMut 作为智能指针，利用 Rust 的 Drop 特质实现自动资源释放
- 当 RTIntrRefMut 超出作用域时，自动恢复中断级别，降低了手动管理资源的风险

3. 灵活的访问方式

- 提供 exclusive_access ()方法获取独占访问权限
- 提供 exclusive_session ()方法在闭包中执行临界区代码
- 提供 as_ptr ()和 as_mut_ptr ()方法获取原始指针，方便底层操作
- 提供 field_ptr ()和 field_mut_ptr ()方法获取结构体字段的原始指针，增强灵活性

在 RusT-Thread 内核中，我们使用这个核心模块来保护那些需要同时被中断服务程序和普通线程访问的关键数据，比如线程控制块(TCB)、调度器状态、定时器列表以及内存管理的内部数据。RTIntrFreeCell<T>的作用就是提供一个“中断锁”，确保在任何时候数据访问都是安全的，这对于保证高并发下系统的数据一致性和整体稳定性至关重要。

4.2. 硬件抽象层

硬件抽象层(hardware)是 RusT-Thread 操作系统在 Cortex-M4 架构下的底层硬件支持模块，主要负责

CPU 端口、上下文切换、异常处理和中断管理等功能，为上层系统提供与硬件紧密相关的基础服务。

硬件抽象层是操作系统的核心组成部分，通过这些模块，RusT-Thread 能够实现高效、可靠的硬件资源管理和任务调度，为上层应用提供稳定的运行环境。

模块内容如下：

```
hardware
├── context.rs    # 线程上下文切换，使用内联汇编实现线程上下文保存与恢复
├── cpuport.rs    # CPU 接口-Cortex-M4，定义了异常栈帧、栈帧、CPU 关机、CPU 重启
├── exception.rs  # 异常处理相关函数
├── irq.rs        # 中断管理模块，提供了中断嵌套计数、钩子设置、使能/禁用中断等功能
└── mod.rs        # 统合上述模块，提供对外接口
```

1. 上下文切换模块(context.rs)

负责线程上下文的切换机制，主要功能包括：

- 线程上下文的保存与恢复
- PendSV 中断处理(实际执行上下文切换)
- 提供 rt_hw_context_switch、rt_hw_context_switch_interrupt 和 rt_hw_context_switch_to 等上下文切换处理函数
- 实现线程间的高效切换，是多线程调度的核心机制

2. CPU 端口模块(cpuport.rs)

提供与 CPU 硬件直接相关的底层支持[3]：

- 定义异常栈帧 ExceptionStackFrame 和栈帧 StackFrame 结构
- 实现线程栈初始化函数 rt_hw_stack_init
- 提供 CPU 关机 rt_hw_cpu_shutdown 和重启 rt_hw_cpu_reset 等功能
- 可选的 FPU 支持(浮点运算单元)

3. 中断管理模块(irq.rs)

实现中断处理相关的功能：

- 中断嵌套计数管理
- 提供中断使能/禁用函数 rt_hw_interrupt_disable/enable
- 中断进入/退出处理 rt_interrupt_enter/leave
- 支持中断钩子函数设置(通过特性开关控制)
- 获取中断嵌套层数 rt_interrupt_get_nest

4. 异常处理模块(exception.rs)

负责处理系统运行中的各种异常：

- 提供硬件错误处理(HardFault、MemManage、BusFault、UsageFault)
- 异常信息收集与输出
- 支持异常钩子机制 rt_hw_exception_install

- 详细的故障跟踪与诊断功能

4.3. 进程调度层

作为 RusT-Thread 操作系统的核心，进程调度层负责管理和调度所有线程，确保它们能够高效、公平地共享处理器资源，实现并发执行，核心职责如下：

- **任务调度**：依据预设调度策略，决定处理器执行权的分配，保障高优先级任务的及时响应，同时兼顾任务的公平执行。支持多种调度算法，具备灵活适应不同应用场景和实时性要求的能力。
- **调度算法选择**：提供优先级调度算法(如优先级 + 时间片轮转)和多级反馈队列调度算法等，用户可根据实际需求灵活选择调度算法，满足不同任务对实时性和资源分配的需求。
- **线程 API 接口**：为用户提供了一系列丰富的接口，使用户能够更加便捷地对线程状态进行调整和控制，例如创建、删除、挂起、恢复线程等操作，我们设计的线程状态转化如图 2 所示，同时提供了获取和设置线程属性的接口，方便用户根据实际需求对线程进行精细化管理。

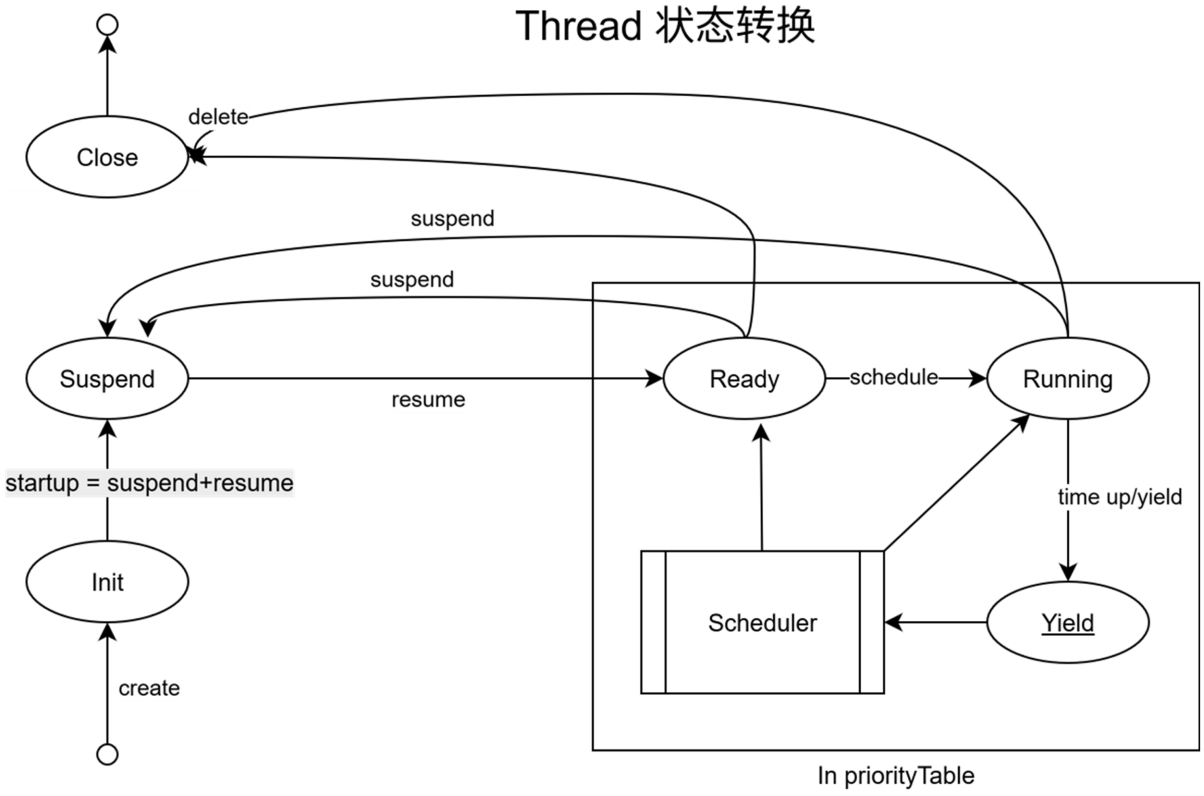

Figure 2. Process state transition
图 2. 进程状态转换图

- **调度策略抽象**：通过定义 SchedulingPolicy 特质，将多种调度算法封装和抽象。以优先级调度算法为例，实现了相应的结构体，包含线程优先级队列管理和时间片分配与轮转机制；对于多级反馈队列调度算法，设计了多级队列数据结构，实现了任务根据执行时间动态调整队列级别的逻辑。基于特质的抽象方式使调度策略切换灵活，用户在初始化时指定调度策略后，调度器即可根据具体实现执行调度操作。

```
/// 调度策略

traitpub trait SchedulingPolicy: Send + Sync {

    /// 选择下一个要运行的线程

    ///

    /// # 参数

    /// * `current_thread` - 当前运行的线程

    ///

    /// # 返回值

    /// * `Option<(Arc<RtThread>, bool)>` - (选中的线程, 是否需要将原线程重新插入就绪队列)

    fn select_next_thread(

        &self,

        current_thread: &Option<Arc<RtThread>>,

    )

    -> Option<(Arc<RtThread>, bool)>;

    /// 获取策略名称

    fn get_policy_name(&self) -> &'static str;

    }
```

- **调度算法优化：**引入位图 + FFS (Find First Set)算法提升调度效率。位图标识各优先级队列的线程就绪状态，通过 FFS 算法快速定位最高优先级队列中的首个就绪线程。FFS 算法借助预计算查找表，将复杂位运算转化为简单数组访问，实现 O (1)时间复杂度的最高优先级线程查找，显著提升调度器响应速度和实时性能，确保系统及时响应高优先级任务。

```
#[cfg(feature = "tiny_ffs")]const __LOWEST_BIT_BITMAP: [u8; 37] = [

    0, 1, 2, 27, 3, 24, 28, 32, 4, 17, 25, 31, 29, 12, 32, 14,

    5, 8, 18, 32, 26, 23, 32, 16, 30, 11, 13, 7, 32, 22, 15, 10,

    6, 21, 9, 20, 19];

#[cfg(feature = "tiny_ffs")]pub fn __rt_ffs(value: u32) -> u8 {

    if value == 0 {

        return 0;

    }

    __LOWEST_BIT_BITMAP[((value & (value - 1) ^ value) % 37) as usize]}
```

该算法利用预计算查找表和数学运算避免复杂位运算循环，实现了快速最低有效位查找。它专为嵌入式环境优化，在资源受限设备上也能快速执行，满足实时系统低延迟要求。能快速确定线程就绪队列中最高优先级线程，确保调度器及时响应高优先级任务，提升系统实时性和性能。

- **多级反馈队列调度实现：**在多级反馈队列调度算法中，引入老化机制，使长期未被调度的线程逐渐升高优先级，防止低优先级线程饥饿，增强调度算法公平性和适应性。实现过程中，借助 Rust 语言零成

本抽象特性，优化代码实现，提升系统稳定性和可维护性。

4.4. 内存管理层

在 RusT-Thread 中，我们实现了原本 RT-Thread 中的小内存分配器，同时并支持了库实现的 buddy_system allocator 和 good_memory allocator，可供用户在实际场景中自由选择，下面将对小内存分配器作具体分析：

RusT-Thread 中的小内存分配器主要体现在如下几个文件中，具体的内存数据组织方式如图 3 所示：

- small_mem_impl.rs：核心算法实现
- small_mem_allocator.rs、allocator.rs：分配器接口与适配
- object.rs、safelist.rs：辅助对象和安全链表
- oom.rs：内存溢出处理

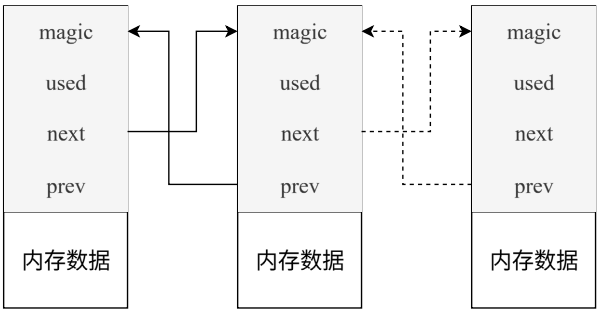

Figure 3. Memory data organization method
图 3. 内存数据组织方式

表示内存块的结构体：

```
///RTSmallMemItem 结构体表示一个小内存块的基本信息，主要用于管理内存池中的单个内存块#[repr(c)]pub struct
RTSmallMemItem {

  ///内存池指针

  pub pool_ptr: usize,

  #[cfg(target_pointer_width = "64")]// 条件编译，64 位系统生效

  pub resv: u32,//保留字段，用于对齐，64 位系统下使用

  ///下一个空闲块的指针

  pub next: usize,

  ///前一个空闲块的指针

  pub prev: usize,

  #[cfg(feature = "mem_trace")]//条件编译，内存跟踪生效

  #[cfg(target_pointer_width = "64")]//条件编译，64 位系统生效

  pub thread: [u8; 8],//线程 ID，64 位系统下使用

  #[cfg(feature = "mem_trace")]//条件编译，内存跟踪生效

  #[cfg(target_pointer_width = "32")]//条件编译，32 位系统生效

  pub thread: [u8; 4],//线程 ID，32 位系统下使用}
```

小内存分配算法原理是通过维护一块连续的内存池，将其划分为带有头部信息的内存块，并用链表管理空闲和已用块。分配时遍历空闲链表，找到足够大的块后分割并标记为已用；释放时将块标记为空闲，并尝试与相邻空闲块合并以减少碎片。整个过程包含边界检查和中断保护，确保分配、释放的安全性和原子性。

除了实现基本的小内存算法外，我们还有如下亮点：

(1) 边界检查与安全性提升

C 代码主要依赖 RT_ASSERT 等宏进行运行时断言，且大量裸指针操作，容易出现悬垂指针、越界、重复释放等问题，这些断言如果被关闭，代码安全性大幅下降。

```
debug_assert!((mem as usize) >= ((*small_mem).heap_ptr as usize));

debug_assert!((mem as usize) < ((*small_mem).heap_end as usize));

debug_assert!(mem_is_used(mem));

if m.is_null() || size == 0 {

    return ptr::null_mut();

}
```

Rust 利用类型系统和所有权机制，天然防止了大部分内存安全问题，同时 rt_smem_free、rt_smem_alloc 等函数在操作前都做了空指针和边界检查。

Rust 的 debug_assert!只在 debug 模式下生效，release 下可关闭，但类型系统和生命周期机制依然提供了额外的安全保障。

许多辅助函数(如 mem_is_used、mem_pool 等)都用 inline 和类型安全的方式实现，减少了手动错误。

(2) 中断保护

C 语言通过 rt_hw_interrupt_disable/rt_hw_interrupt_enable 手动保护关键区，防止并发破坏堆结构。

```
rt_base_t level = rt_hw_interrupt_disable();

// ...关键区.

rt_hw_interrupt_enable(level);
```

Rust 同样调用 rt_hw_interrupt_disable/rt_hw_interrupt_enable，但更易于用 RAII (资源自动释放)等机制进行封装，减少人为失误。

```
let level = rt_hw_interrupt_disable();

//...关键区 ...

rt_hw_interrupt_enable(level);
```

并且 Rust 代码结构更清晰，便于后续用 RAII 或作用域自动恢复中断，提升健壮性。

4.5. 线程通信层

进程间通信(IPC)是多任务操作系统中各个任务之间进行数据交换和协同工作的重要手段。我们的 RusT-Thread 提供了信号量机制，而消息队列、邮箱等作为拓展，我们尚未支持。

信号量工作示意图如图 4 所示，每个信号量对象都有一个信号量值和一个线程等待队列，通过信号量的值是否为零，决定线程是否可以访问临界区的资源，当信号量实例数目为零时，再申请该信号量的线程就会被挂起在该信号量的等待队列上，等待可用的资源。

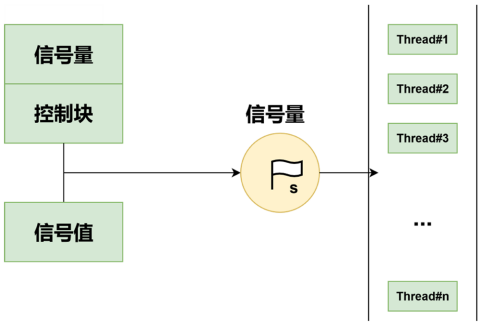

Figure 4. Semaphore working diagram
图 4. 信号量工作示意图

在 RT-Thread 中，信号量相关操作有以下函数——创建、删除、获取、释放，我们实现的思路和 C 类似，先实现 ipc 的基础操作，如_ipc_list_suspend 挂起线程，_ipc_list_resume 唤醒线程等等，ipc 相关操作函数如图 5 所示，然后，通过信号量的值，选择不同的操作，实现信号量的相关操作即可。

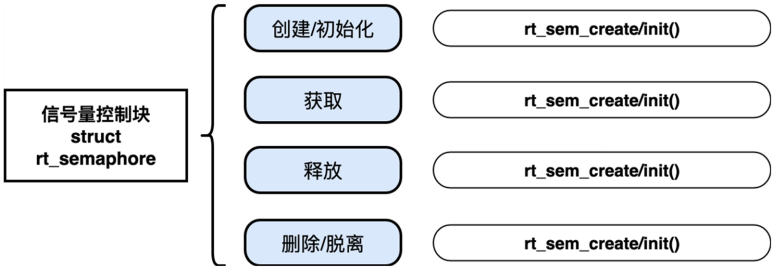

Figure 5. Semaphore related operation functions
图 5. 信号量相关操作函数

4.6. 时钟控制层

4.6.1. 时钟节拍的产生

时钟节拍由配置为中断触发模式的硬件定时器产生，当中断到来时，将调用一次 rt_tick_increase ()函数，通知操作系统已经过去一个系统时钟；不同硬件定时器中断实现都不同，Rust_Thread 的中断函数是在 QEMU 模拟器上的 stm32 系列单片机上实现的，具体地，程序中将使用#[exception]一个中断处理函数 SysTick ()，在其中调用 rt_tick_increase ()函数。

在 RusT-Thread 系统中，使用常数 RT_TICK_PER_SECOND 控制时钟周期长度。本系统中默认主频是 16 MHZ，是通过 QEMU 模拟芯片内部高速振荡器实现的，默认 RT_TICK_PER_SECOND = 1000，即一个时钟周期 16000 个硬件周期。

4.6.2. 时钟中断的管理

时钟中断管理核心函数 rt_tick_increase ()主要完成以下工作：

- 将全局变量 RT_TICK 自增，这个变量记录了系统从初始化到当前经过了多少个时钟周期，叫做系统时间。

- 检查当前线程的时间片是否到期，若到期，则触发线程调度。
- 检查是否有定时器到期，如果有，触发定时器超时函数。

时钟管理中还包括系统时间的读取和设定函数，毫秒数和时钟周期数的转换函数等功能函数。

4.6.3. 定时器的管理

RT-Thread 的定时器由定时器控制块 RtTimer 控制，定时器控制块全部在运行时动态分配内存并按照超时时间升序挂载在动态数组 TIMERS 中。定时器的管理主要包括定时器的创建、激活、修改、超时与停止。

- 创建：使用 new 方法创建，体现了面向对象的思想。
- 激活：计算超时时刻，通过二分查找插入 TIMERS 数组，保证有序性。
- 修改：使用 enum 封装所有修改操作(如周期、定时值)，符合 Rust 风格且易于扩展。修改在下次激活时生效。
- 超时：通过二分查找高效定位到期的定时器，执行回调。周期性定时器会自动重新激活，单次定时器则被移除。
- 停止：从 TIMERS 数组中移除并回收定时器。

5. RusT-Thread 性能与验证测试

5.1. 内存性能测试

原生的 RT-Thread 官方并没有给出一些具体的有关内存性能的数据，所以这里我们参照标准的 std 库来比较分析 RusT-Thread 的性能。

我们在 Linux (x86_64)平台下，使用 Rust 重新实现了 RusT-Thread 的完全相同的内存管理模块，并对其进行了适当的封装，并使其接口与原系统保持一致。这样，我们就可以在支持 std 库和 Criterion 基准测试框架的环境下，对内存分配、释放等核心操作进行高效、可重复的性能测试，并与 Rust 标准库分配器进行公平对比。

- 测试代码使用 Criterion 框架实现，测试内容包括小块分配、混合分配、碎片处理、内存利用率等多种典型场景。
- 这种方法虽然不能完全反映嵌入式平台的绝对性能，但可以有效比较不同分配算法的相对性能，为实际部署和优化提供有价值的参考。

具体测试结果如图 6 所示。

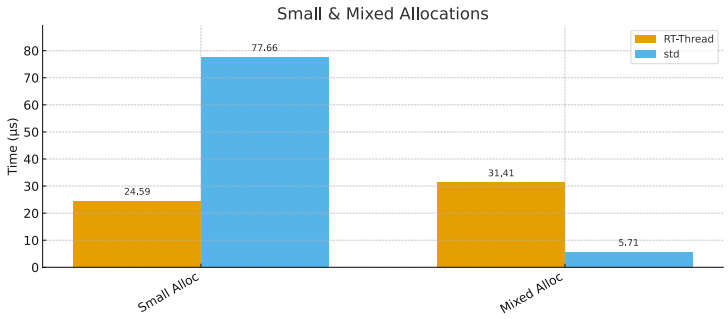

Figure 6. Memory performance test results
图 6. 内存性能测试结果

图 6 展示了 RusT-Thread 与标准分配器在进行小块内存分配和混合内存分配时的性能表现：

- Small Allocations 中，RusT-Thread 平均耗时约为 24.6 μs，明显优于 std 的 77.7 μs，表明 RusT-Thread 在频繁的小对象分配中具有更低的管理开销。
- Mixed Allocations 中，标准分配器表现优异，耗时仅约 5.7 μs，而 RusT-Thread 则为 31.4 μs，可能是由于 RusT-Thread 对变长内存块的处理不如标准库灵活高效。

RusT-Thread 在固定小块内存操作中具有优势，但在面对内存尺寸变化复杂的情况时性能下降。不过这同时也展现出不同应用场景中，RusT-Thread 中小内存分配算法的性能更加稳定。

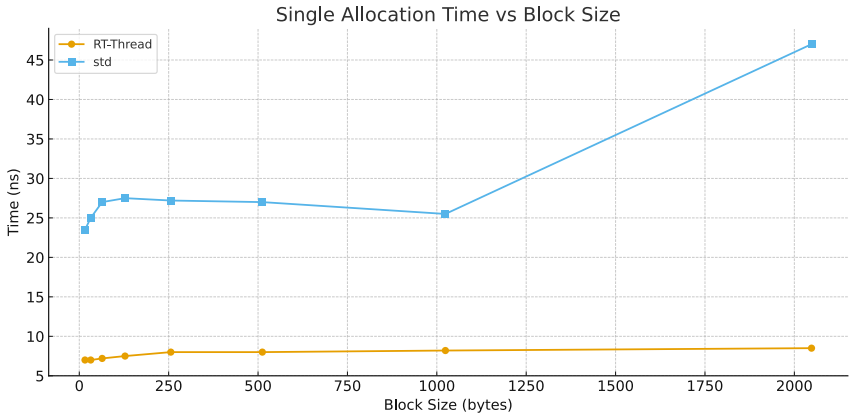

Figure 7. Trend of single allocation time changes for RusT-Thread and standard allocator
图 7. RusT-Thread 和标准分配器的单次分配耗时变化趋势

图 7 展示了在逐步增加内存块大小的条件下，RusT-Thread 和标准分配器的单次分配耗时变化趋势：

- RusT-Thread 的分配时间基本稳定在 7~8 ns 范围内，说明其设计对小中等大小块分配进行了优化处理，性能几乎不受块大小影响
- 标准分配器的耗时随着块大小增加而变化更为剧烈，例如从 16B 的约 24 ns 上升到 2048B 时超过 47 ns，说明其可能使用了更复杂的分配策略或存在内存对齐开销

可见，RusT-Thread 分配器的响应速度更稳定，适用于内存块尺寸变化不大的嵌入式任务场景。

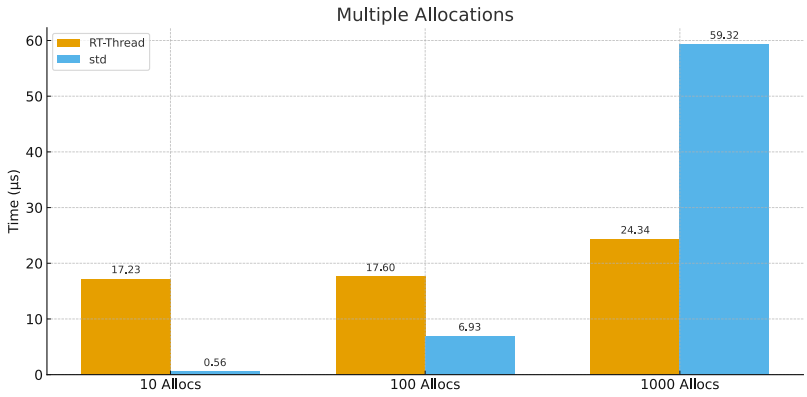

Figure 8. Average time taken by RusT-Thread allocator and standard allocator for bulk memory allocation
图 8. RusT-Thread 分配器与标准分配器在进行批量内存分配时的平均耗时

图 8 对比了在进行批量内存分配(10、100、1000 次)时，两种分配器的平均耗时：

- RusT-Thread 在 10~100 次批量分配中仅需 17 μs 左右，而标准分配器耗时逐步上升到 60 μs 以上。
- 差距在批量操作中进一步放大，表明 RusT-Thread 的批处理性能更优，内部结构对频繁申请释放有较强的适应性。

可以看出，RusT-Thread 分配器在多次重复分配的密集型任务中更具优势，特别适用于任务频繁上下文切换或数据缓冲场景。

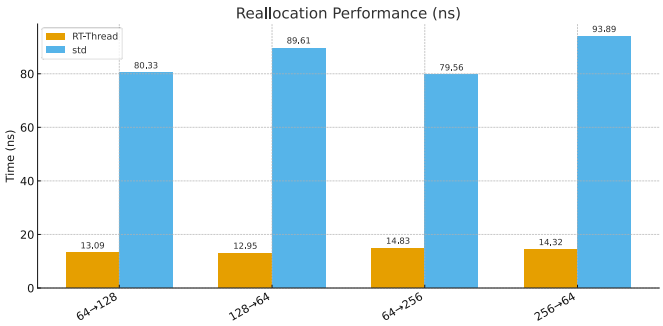

Figure 9. Time comparison under four memory reallocation paths
图 9. 四种内存重分配路径下的耗时比较

图 9 展示了常见四种内存重分配路径(如 64→128，128→64，64→256，256→64)下的耗时比较：

- 所有场景中 RusT-Thread 的耗时都远低于标准分配器，例如 128→64 仅约 12.95 ns，而标准库约 89.6 ns
- 表明 RusT-Thread 分配器可能采用就地扩展或快速搬迁机制，避免了额外的复制和元信息更新开销

所以，RusT-Thread 对 reallocation 操作进行了优化，在需要动态改变内存块大小的场景中表现更加出色，这使得对于真实环境中的复杂情况的处理会更加优秀。

5.2. 模块验证测试

测试部分代码结构如下：

```
RusT-thread/src/test
├── README.md                 # 测试/示例的说明与使用方法，概览覆盖点与调用入口
├── comprehensive_example.rs  # 全面功能演示（线程/定时器/调度策略/MFQ/中断等全量展示）
├── example.rs                # 基础功能演示入口（线程生命周期、优先级/睡眠/让出等）
├── example_mfq.rs            # 多级反馈队列（MFQ）调度策略的示例与演示。
├── integration_test.rs       # 系统级集成测试套件，覆盖线程/内存/调度/定时器/IPC，并发、压力与边界场景。
├── mod.rs                    # 测试模块聚合与导出，集中开关
├── performance_test.rs       # 性能与吞吐/时延评估，随机负载仿真、指标统计与可视化输出。
├── switch_time_test.rs       # 关注上下文切换/调度路径时间测量与分析。
├── test_all.rs               # 打印测试 Banner/Logo 的入口展示函数。
├── test_cell.rs              # 中断安全数据结构（如 RTIntrFreeCell）的并发访问测试。
├── test_excp.rs              # 异常处理相关路径的触发与校验。
├── test_interupt.rs          # 中断启停与相关功能的基础测试。
├── test_ipc.rs               # IPC 队列挂起/唤醒/全唤醒与优先级队列语义的多线程联调。
├── test_mem.rs               # 基础内存分配/释放，Vec/Box 等简单路径验证。
├── test_scheduler.rs         # 调度器插入/启动/调度行为的功能性测试。
├── test_small_mem.rs         # 小块内存分配器的专项测试（受 feature 控制）。
├── test_thread.rs            # 线程基本能力测试（创建/启动/挂起/恢复）与裸上下文切换验证。
└── test_timer.rs             # 定时器单次/周期回调与停止控制的功能测试。
```

5.3. 单元测试

我们为各个模块编写了详细的单元测试用例，对模块的功能进行充分验证。例如，对线程管理模块，测试了线程的创建、启动、挂起、恢复、删除等基本操作，以及不同调度策略下的线程切换和优先级管理；对内存管理模块，测试了内存的分配、释放、重分配等操作以及对标准 alloc 的容器支持；对定时器模块，测试了定时器的创建、启动、停止、重启等操作，以及定时器回调函数的执行等。

5.4. 集成测试

在模块间集成测试中，我们重点关注各模块协作及系统整体功能正确性，设计了以下典型测试场景：

1. 并发性测试

- 并发线程创建：多线程同时创建 RT-Thread 线程，验证线程管理安全性。
- 优先级调度验证：创建不同优先级线程并记录执行顺序，确保调度器优先级语义正确。
- 内存碎片化测试：模拟复杂内存分配释放场景，验证内存管理器健壮性。

2. 压力测试

- 大规模线程创建：创建 100 个线程同时运行，测试系统高负载稳定性。
- 内存分配压力测试：进行 1000 次随机大小内存分配释放，验证内存管理器性能。
- 定时器密集测试：创建 50 个定时器同时运行，测试时钟系统处理能力。

3. 边界条件和错误处理

- 边界值测试：测试最小/最大优先级、最小内存分配、零大小分配等边界情况。
- 错误场景模拟：测试重复启动线程、重复挂起线程、无效线程 ID 等异常处理。
- 资源耗尽测试：模拟内存不足等资源耗尽场景，验证系统错误处理机制。

4. 系统稳定性测试

- 长时间运行测试：验证系统长期稳定性。

5.5. 性能基准测试

为验证 RusT-Thread 的实时性能，我们设计了针对 RTOS 核心指标的性能基准测试体系。测试重点关注时间确定性和系统响应的可预测性，这是实时操作系统区别于通用操作系统的关键特征。我们选择了四项核心指标进行评估：中断延时测试验证系统对外部事件的响应速度，响应时间测试评估任务调度的实时性，上下文切换时间 测试衡量线程切换效率，线程启动时间测试检验系统资源分配性能。这些测试不仅帮助我们发现性能瓶颈和优化空间，更重要的是为系统的实时性保证提供了量化依据，确保 RusT-Thread 能够满足嵌入式和工业控制应用的严格时间要求[4] [5]。

5.5.1. 中断延时测试

我们利用 Cortex-M 内核自带的 SysTick (系统滴答定时器)来精确测量中断延时。SysTick 本质上是一个 24 位的硬件自减计数器。其工作原理如下：

- **计数与触发：**计数器以系统时钟频率从一个预设的重载值(Reload Value)开始递减。当计数值从 1 减到 0 时，硬件会立即触发 SysTick 中断。
- **自动重载：**在触发中断的同一时钟周期，硬件会自动将重载值重新加载到计数器中，使其无缝开始下一轮递减，整个过程无需任何软件干预。

由于中断服务程序的执行会存在微小的延迟，而 SysTick 计数器在此期间并未停止。因此，我们可以在中断服务程序的入口处，立即读取此刻 SysTick 计数器的当前值。通过这个差值，我们就能精确计算出

中断延时。计算公式为：

$$\text{中断延时} = (\text{重载值} - \text{当前值})/\text{系统时钟频率}$$

我们系统测得 1000 次平均中断延时为 1.21 us。

5.5.2. 响应时间测试

事件响应时间是指从事件发生到系统完成相应处理的总时间，包括事件检测、任务调度和执行处理逻辑的全过程。此指标衡量了系统对外部事件的端到端处理能力，是评估实时操作系统性能的综合指标。

在我们的测试中，使用随机数生成器模拟事件的随机生成，每隔相同时间生成一个随机数并与一个给定的概率值比较，当小于此概率值时，就生成一个事件，此事件的优先级也为随机数，可以证明，事件的间隔服从泊松分布。

我们将事件分为三种优先级：高中低，同时创建三个不同优先级的处理程序来处理事件，测量平均响应时间和各优先级的响应时间，结果如图 10 所示。

Figure 10. Event response time test results
图 10. 事件响应时间测试结果

一般硬实时操作系统响应时间的指标如表 1 所示。

Table 1. Hard real-time operating system response time indicators
表 1. 硬实时操作系统响应时间指标

优先级	高	中	低
响应时间(us)	1~5	5~20	20~100

可以看到我们三种优先级事件的响应时间均符合硬实时操作系统的要求。

5.5.3. 上下文切换时间测试

上下文切换时间是指操作系统从一个线程切换到另一个线程所需的时间，包括保存当前线程状态和恢复目标线程状态的过程。该指标对多任务系统的整体性能有显著影响，它决定了系统在任务间切换的效率。上下文切换时间越短，系统在高负载下的响应性越好，尤其在资源受限的嵌入式系统中，高效的上下文切换能显著提高处理器利用率和系统吞吐量。

我们测试了两个相同优先级线程来回切换 5000 次的平均时间，并执行了 100 次这样的测试，求得线程切换的时间性能如表 2 所示。

Table 2. Time performance of thread switching
表 2. 线程切换的时间性能

测试项目	结果
总测试次数	100
平均切换时间	1.75 μs
最小切换时间	1.11 μs
最大切换时间	2.58 μs
切换时间标准差	0.31 μs

5.5.4. 线程创建时间测试

线程创建时间是指从发起创建线程请求到新线程可以被调度执行所需的时间。该指标反映了系统动态资源分配和任务管理的效率。在需要频繁创建临时任务的应用场景中(如 Web 服务器、动态负载系统)，高效的线程创建机制可以显著降低系统开销，提高资源利用率。对于嵌入式实时系统，快速的线程创建能力也有助于系统在运行时更灵活地调整工作负载，适应变化的环境需求。

由于 flash 大小限制，单次测试创建 50 个线程的平均时间，再做 100 次测试求平均，得到测试结果如表 3 所示。

Table 3. Average thread creation time
表 3. 线程平均创建时间

测试项目	结果
总测试次数	100
平均创建时间	2.69 μs

6. RusT-Thread 与 RT-Thread 性能对比

RT-Thread 官方给出了他们的性能测试结果如图 11 [6]。

RT-Thread性能指标

RT-Thread 实时内核基于 Zynq7020平台测试的性能指标数据如下：

中断延时

系统心跳时钟频率 1000 Hz，测试总样本数 100000，99.59% 的中断延时数据分布在 280 ns 到360 ns 之间。

	最小值(ns)	平均值(ns)	最大值(ns)
中断延时时间	162	321	948

上下文切换指标

功能	最小值(us)	平均值(us)	最大值(us)
Thread switch by mailbox	0.801	0.854	1.399
Thread switch by semaphor	0.711	0.762	1.405
Thread switch by suspend	0.597	0.633	1.177

线程间通信指标

功能	最小值(us)	平均值(us)	最大值(us)
MBox send time	0.123	0.135	0.525
MBox receive time	0.135	0.136	0.228
MemPool allocate time	0.105	0.117	0.519
MemPool free time	0.093	0.099	0.249
Semaphore take time	0.099	0.111	0.222
Semaphore release time	0.093	0.099	0.138

线程创建指标

功能	最小值(us)	平均值(us)	最大值(us)
Thread create time	2.696	2.969	3.579
Thread init time	2.132	2.402	3.474

Figure 11. RT-Thread official performance test results
图 11. RT-Thread 官方性能测试结果

其测试基于的硬件平台是 Zynq 7020，该开发板的主频为 800 MHz，而我们使用的 QEMU 模拟的是 stm32f405 的开发板，主频为 168 MHz，在对比性能时应考虑相关的硬件资源。

Table 4. Performance comparison between RusT-Thread and RT-Thread
表 4. RusT-Thread 与 RT-Thread 性能对比

指标	RT-Thread	RusT-Thread	折合后的等效时间
中断延时(ns)	321	1210	254
上下文切换(us)	0.633	2.58	0.542
线程创建(us)	2.969	2.60	0.546

综上，我们重写后的 RusT-Thread 操作系统的实时性与 RT-Thread 相当甚至更加优秀(表 4)，这符合我们当初制定的性能提升的目标。

7. RusT-Thread 优点和缺点

7.1. 优点

1. 安全导向的设计语言。Rust 的所有权机制使得 RusT-Thread 操作系统在内存安全和并发安全上很有优势[7] [8]。

2. 高度模块化并且可以定制。在搭建 RusT-Thread 操作系统时，我们基于 Rust 语言特性，建立了可模块化的系统仓库，定义改写模块方便快捷。

3. 平台具备高度可扩展性，虽然目前仅支持 Cortex-M4 平台，但模块化的 Hardware 利于后续扩展平台。

4. 多样的算法特性选择。在线程调度上提供多种调度算法，如优先级、优先级 + RR 等；在内存管

理上支持伙伴系统、小内存分配器等多种内存分配方式。

5. 性能优秀。在 Rust 改写后的系统线程上下文切换性能与原 C 程序性能相当。

6. 测试点丰富。我们对各个模块开发了功能测试程序，利于后续的调试开发。

7.2. 缺点

1. 相关文档尚不完善。由于时间精力，我们暂未维护起完善的文档体系。

2. Rust 所有权机制等致使代码可读性差。为开发带来一定困难。

3. 外设周边尚未开发支持。

4. 异常反馈系统尚不完善，待后续开发。

8. 总结与展望

历经数月的攻坚克难，RusT-Thread 项目终于迎来了阶段性成果。我们成功地将 RT-Thread nano 内核的核心功能，包括线程调度、内存管理、中断处理、时钟服务、IPC 等，用 Rust 语言进行了高质量的重构与实现。这不仅是一次技术栈的迁移，更是在嵌入式实时操作系统领域，对 Rust 语言安全性、并发性和现代语言特性的一次深度实践与验证。

8.1. 核心成果：安全性与性能的双重提升

1. 内存安全基石

最大的收获莫过于 Rust 强大的所有权系统和借用检查机制带来的根本性改变。通过 RTIntrFreeCell 等创新设计，我们有效解决了嵌入式系统中全局共享数据访问这一高危痛点，将数据竞争、野指针、缓冲区溢出等 C 时代常见的“幽灵”从编译期就扼杀在摇篮里。内核服务的错误处理也因 Option/Result 变得更加健壮和可预测。

2. 性能不妥协

我们并非单纯追求“安全”而牺牲效率。精心优化的调度算法(如基于位图 + FFS 的优先级调度、创新的多级反馈队列)、高效的小内存分配器实现，以及 Rust 零成本抽象的特性，共同确保了 RusT-Thread 在 QEMU 模拟环境下的性能指标(中断延时、上下文切换、线程创建、响应时间)与原版 C 实现的 RT-Thread Nano 相当甚至略有优势。这证明了 Rust 完全有能力胜任对实时性要求苛刻的嵌入式场景。

3. 代码精简与清晰

Rust 的现代语言特性(如 trait、泛型、丰富的标准库容器)显著提升了代码的表达力和可维护性。最直观的体现是，在实现同等甚至更多功能(如更优的定时器二分查找算法)的前提下，RusT-Thread 的核心代码量(约 5500 行)相比原 C 版(约 9600 行)有了显著的精简。模块化设计和清晰的抽象也让代码结构更易于理解和扩展[9]。

4. 扎实的验证体系

我们构建了涵盖单元测试、集成测试和全面的性能基准测试(内存性能、中断延时、响应时间、上下文切换、线程创建)的验证体系。详实的数据不仅证明了系统的功能正确性，也为性能优化和后续迭代提供了坚实基础。

8.2. 挑战与突破：在“裸机”上驾驭 Rust

项目过程并非一帆风顺。调试手段匮乏时，我们深度依赖半宿主打印和内联汇编调试技巧；硬件对接和启动流程的复杂性，通过 cortex-m、cortex-m-rt 等库结合内联汇编巧妙化解；汇编与 Rust 联合编译

调试的难题，最终以内联汇编统一在 Rust 源码中的方案优化解决[8]；而困扰嵌入式开发的全局变量问题，则由 RTIntrFreeCell + lazy_static 的组合拳提供了安全可靠的 Rust 式解决方案。每一次挑战的克服，都加深了我们对 Rust 在嵌入式裸机环境应用的理解。

8.3. 展望未来：构建更强大、更开放的 RusT-Thread 生态

RusT-Thread 的诞生只是一个起点，我们对其未来充满期待：

1. 功能深化与扩展

(1) 丰富软件生态

系统性地移植 RT-Thread 社区成熟的核心软件包(网络协议栈 lwIP/PicoTCP、文件系统 LittleFS/SPIFFS、GUI 组件等)，是当务之急。我们将致力于构建标准化的 Rust-C 互操作层接口，让海量的现有 C 语言资源能更顺畅地融入 Rust 生态。

(2) 高级内核特性

实现更完善的内存管理策略(Slab, MemHeap)、支持更丰富的 IPC 机制(消息队列、邮箱、事件集)、探索多核(SMP)支持将是内核层面的重要方向。

(3) 人性化体验

当前错误处理主要透传底层错误码。未来计划引入结构化的 Rust-native 错误类型，并集成分级日志与堆栈追踪功能，让异常反馈更清晰、调试更高效。

2. 生态建设与普及

(1) 广泛的硬件支持

目前已在 Cortex-M4 上验证。下一步将适配更多主流架构如 Cortex- M0+/M3/M7 和 RISC-V，目标是覆盖更广泛的物联网和边缘计算硬件平台。

(2) 清晰的开发者体验

完善多级文档体系是生态繁荣的关键：

- 代码级：维护详尽的 rustdoc API 文档，包含示例和安全性说明。
- 模块级：编写硬件抽象层(HAL)指南、驱动移植教程、核心模块设计解析等。
- 入门级：提供面向应用开发者的、易于上手的使用手册和丰富的示例项目，显著降低 Rust 嵌入式开发的门槛。

(3) 工具链优化

持续优化代码体积(如替换重型打印宏)、提升构建体验，并探索更好的调试支持集成(如更深入的 GDB 支持)。

RusT-Thread 项目是一次勇敢的尝试，它证明了 Rust 在资源受限的实时操作系统领域不仅可行，更能带来显著的安全性和开发效率提升。我们重构的不仅是一套代码，更是在探索嵌入式系统开发的未来范式。代码已开源，这只是一个开始。我们热切期待更多对 Rust 和嵌入式系统感兴趣的开发者加入，共同打磨 RusT-Thread，将其打造成为一个真正安全、高效、易用的开源实时操作系统选择，为国产嵌入式基础软件生态注入新的活力！安全至上，性能无忧，Rust 让嵌入式未来更可期。

致　谢

本项目来自于中国科学技术大学操作系统 H 课大作业，在这里感谢任课老师邢凯对本项目的指导，感谢课程助教团队对本项目的帮助！

参考文献

[1] RT-Thread 文档中心[EB/OL]. https://www.rt-thread.org/document/site/#/, 2025-04-21.

[2] Klabnik, S. and Nichols, C. (2018) The Rust Programming Language. No Starch Press.

[3] 陈渝, 尹霞, 张峰. Rust 语言机制与安全性[C]//第 39 次全国计算机安全学术交流会论文集. 2024.

[4] 胡霜, 华保健, 欧阳婉容, 樊淇梁. 语言安全研究综述[J]. 信息安全学报, 2023,, 8(6): 64-83.

[5] Criterion 测试工具[EB/OL]. https://docs.rs/criterion/latest/criterion/, 2025-05-23.

[6] Pompeii, E. (2024) How to Benchmark Rust Code with Criterion. https://bencher.dev/learn/benchmarking/rust/criterion/

[7] 千锋教育第 3 章 RT-Thread 内核介绍[EB/OL]. https://zhuanlan.zhihu.com/p/641915283, 2025-04-21.

[8] 乐鑫科技 Rust + 嵌入式: 强力开发组合[EB/OL]. https://zhuanlan.zhihu.com/p/628575325, 2025-04-21.

[9] RT-Thread 产品性能[EB/OL]. https://www.rt-thread.com/products/Performance-38.html, 2025-04-21.