Research Report

研究背景

RTOS 简介

实时操作系统 (RTOS) 是一种体积小巧、确定性强的计算机操作系统。 RTOS 通常用于需要在严格时间限制内对外部事件做出反应的嵌入式系统，如医疗设备和汽车电子控制单元 (ECU)。 通常，此类嵌入式系统中只有一两项功能需要确定性时序，即使嵌入式系统不需要严格的实时反应，使用 RTOS 仍能提供诸多优势。支持多任务处理与并发，调度与实时调度等操作系统功能。由于 RTOS 确保了系统的实时性和可靠性,所以更应该用在对时间要求有限制的嵌入式设备中(如工业控制系统,家用电器系统,物联网系统等)

FreeRTOS 简介

FreeRTOS 是一个轻量级的实时操作系统，其功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需要。
其特点（粗略）有：

- 用户可配置内核功能
- 目标代码小，简单易用
  核心代码只有**3到4个**C文件，占用了较少的系统资源，是为数不多能够在小RAM单片机上运行的RTOS; 大部分的代码都是以**C语言编写**，只有一些函数采用汇编语言编写，拥有较强的可读性
- 强大的执行跟踪功能

- 堆栈溢出检测
- 没有限制的任务数量
- 没有限制的任务优先级
- 多个任务可以分配相同的优先权
- 队列，二进制信号量，计数信号灯和递归通信和同步的任务
- 优先级继承
- 免费开源的源代码

FatFs 简介

FATFS 是一个完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。完全用标准 C 语言编写，所以具有良好的硬件平台独立性。可以移植到 8051、PIC、AVR、SH、Z80、H8、ARM 等系列单片机上而只需做简单的修改。它支持 FAT12、FAT16 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读／写，并特别对 8 位单片机和 16 位单片机做了优化。

FATFS 是可裁剪的文件系统(专门为小型嵌入式设计)。有以下特点：

- Windows 兼容的 FAT 文件系统（支持 FAT12 / FAT16 / FAT32）
- 与平台无关，移植简单。全 C 语言编写。
- 代码量少、效率高。
- 多种配置选项
  - 支持多卷（物理驱动器或分区，最多 10 个卷）
  - 多个 ANSI / OEM 代码页包括 DBCS
  - 支持长文件名、ANSI / OEM 或 Unicode
  - 支持 RTOS
  - 支持多种扇区大小
  - 只读、最小化的 API 和 I / O 缓冲区等

Rust简介

Rust是一种现代的系统编程语言，它注重性能、安全性和并发。Rust最初由Mozilla研究院的Graydon Hoare于2006年开始设计，最早的目标是为了解决C++在系统编程领域的一些痛点。在2010年，Mozilla正式开始支持这个项目，从那时起，Rust开始迅速发展并逐渐成为一个强大的编程语言。

Rust的设计目标是为程序员提供一种高性能、安全且具有现代化特性的系统编程语言。以下是Rust的一些主要优势：

内存安全性

Rust通过其独特的所有权系统、生命周期和借用检查器确保了内存安全。这些功能让Rust能够在编译时检测许多常见的内存错误，如悬垂指针、空指针解引用、数据竞争等。这种设计减少了内存泄漏、悬垂指针等问题的出现，从而使得编写安全的代码变得更加容易。

高性能

Rust注重零开销抽象（zero-cost abstractions），这意味着Rust提供的高级抽象不会对程序性能产生负面影响。Rust的性能与C和C++相当，这使得它成为一个理想的选择，尤其是对于对性能要求较高的系统编程任务。

并发友好

Rust的内存模型和类型系统让并发变得更加简单和安全。通过提供原子操作和线程安全的数据结构，Rust在编译时就可以预防数据竞争等多线程问题。这使得Rust在多核处理器和分布式系统领域具有优势。

易于集成

Rust的C兼容的FFI（Foreign Function Interface）允许轻松地与其他编程语言集成。这使得Rust可以逐步替换现有的C和C++代码，提高系统的安全性和性能，而无需重写整个项目。

生产力与现代化特性

Rust提供了许多现代编程语言的特性，如模式匹配、类型推断、闭包等。这些特性可以提高程序员的生产力，使得编写代码更加愉快。Rust的丰富的标准库和第三方库也使得开发者能够轻松地找到所需的功能。

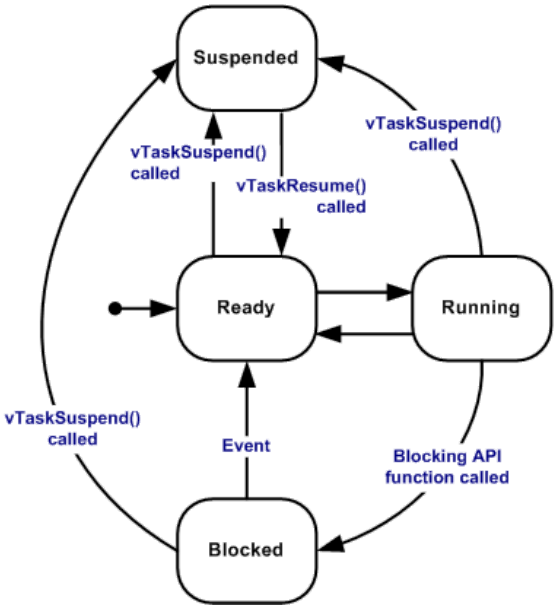
相关调研

FreeRTOS的工作原理

任务

- **任务状态**
  - 运行
  - 准备就绪
    同等或更高优先级的不同任务已经处于运行状态,此任务准备好了执行但没有执行
  - 阻塞
    正在等待时间或外部事件. Eg. 如果一个任务调用vTaskDelay()，它将被阻塞（被置于阻塞状态）， 直到延迟结束
    阻塞状态的任务通常有一个"超时"期， 超时后任务将被超时，并被解除阻塞， 即使该任务所等待的事件没有发生。“阻塞”状态下的任务不使用任何处理时间，**不能** 被选择进入运行状态

- 挂起
  “挂起”状态下的任务不能 被选择进入运行状态，但处于挂起状态的任务 没有超时, 只有在分别通过 vTaskSuspend() 和 xTaskResume() API 调用明确命令时 才会进入或退出挂起状态
- 有效任务转换示意图



- **任务优先级**
  - 每个任务均有优先级
  - 优先级从0到configMAX_PRIORITIES - 1
  - 数字越小表示优先级越低, 空闲任务(IDLE Task)优先级为0
  - 调度器确保处于运行状态的任务的优先级始终是最高的那个
  - If configUSE_TIME_SLICING = 1,则相同优先级的任务采用**时间片轮询**调度方案共享可用的处理时间
- **任务调度**
  - 默认采用**固定优先级的抢占式调度**策略,对同等优先级的任务执行时间切片轮询调度
    - 固定优先级是一个概念,文档里有
    - 抢占式就是优先级最高的会抢占
    - 轮询
    - 时间切片

队列、互斥锁与信号量

- 任务间通信的主要形式。使用队列发送队列接收两个函数来进行通信，新数据可以被发送到队列的后面，也可以发送到前面。
- FreeRTOS队列所使用的模型中，消息通过队列以副本的方式发送，即**数据（可以是更大的缓冲区的指针）本身被复制到队列中， 而不是队列始终只存储对数据的引用。**

- 阻塞队列
  - 任务从空队列中读取或写入一个满队列时，都会让队列进入阻塞状态
  - 如果队列中有多个阻塞状态任务，高优先级的先解除阻塞
- 二进制信号量
  - FreeRTOS二进制信号量是一种长度为1的队列（队列只能满或空），用于任务间或任务与中断间的同步。相比互斥锁，它无优先级继承机制，更适合同步场景。任务通过阻塞获取信号量，中断通过 `xSemaphoreGiveFromISR()` 释放信号量，形成"延迟中断"模式。其轻量级特性可用任务通知替代，但需注意中断安全API的使用限制。
- 计数信号量
  - 二进制信号量可以被认为是长度为 1 的队列，计数信号量也可以被认为是长度大于 1 的队列。同样，**信号量的使用者对存储在队列中的数据并不感兴趣，他们只关心队列是否为空**。
- 互斥锁
  - 包含优先级继承机制的二进制信号量 。互斥锁更适合实现简单的 相互排斥（即互斥）。
  - 用于互斥时， 互斥锁就像用于保护资源的令牌。当一个 任务希望访问资源时，必须首先获得（“获取”）该令牌。使用完资源后， 任务必须“返还”令牌，以便其他任务有机会访问 相同的资源。
- 递归互斥锁
  - 锁几次，就需要解开几次

直达任务通知

- RTOS任务自带有一个任务通知数组，每个任务都有挂起和非挂起的通知状态和一个32为位的通知值。
- 直达任务通知是直接发送到任务的事件，而不是通过中间对象间接发送。发送通知时可选择：
  - 覆盖数值
  - 设置指定位
  - 对值进行增量
- 任务可以阻塞等待特定通知
- 一些API：
  - 发送：xTaskNotifyIndexed () /xTaskNotifyGiveIndexed ()
  - 接收：xTaskNotifyWaitIndexed () /ulTaskNotifyTakeIndexed ()

流缓冲区和消息缓冲区

- 流缓冲区

简单来讲就是一种通信原语

流缓冲区是服务于 任务 -> 任务 或 中断 -> 任务的, 简单来讲是发送者把数据复制到缓冲区,然后接收者从缓冲区赋值数据, 流缓冲区传递连续的字节流

- 消息缓冲区

消息缓冲区传递大小可变但离散的消息, 消息缓冲区使用流缓冲区进行数据传输

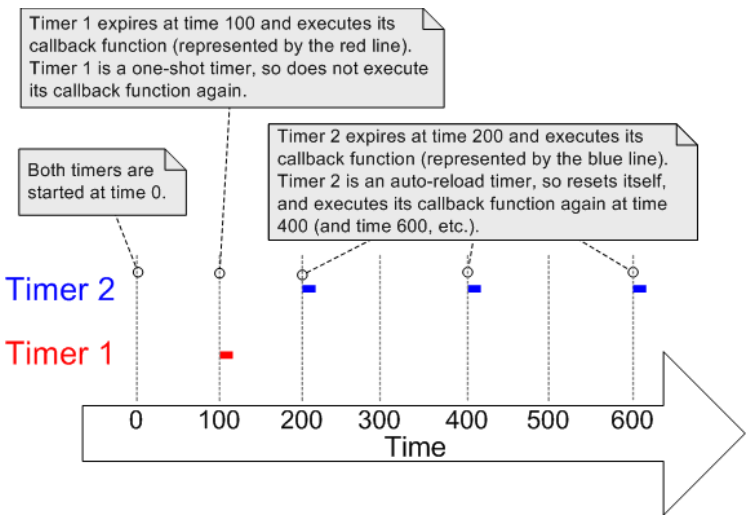
- 这两个缓冲区的实现都保证 发送者和接收者都只能是一个, 如果有多个, 则将发送|读取的API放在临界区(?我不懂,先保留,不是目前的研究重点)

软件定时器

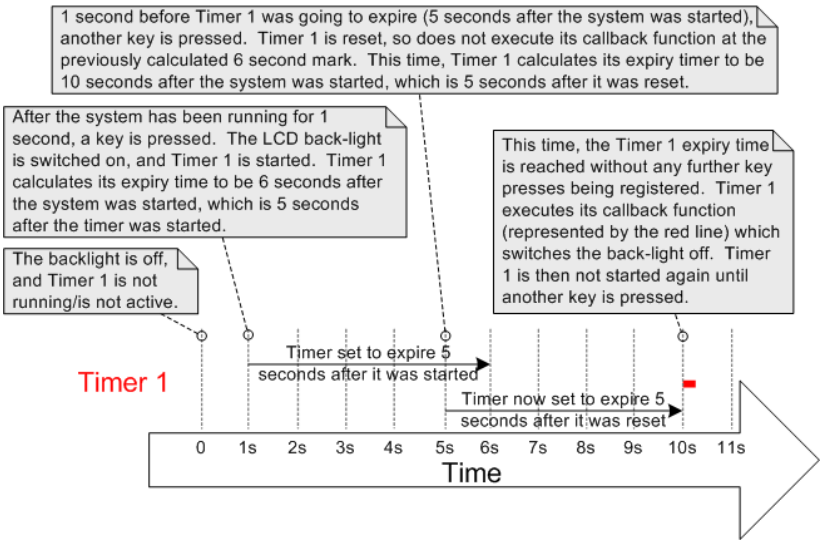
- 让函数在 未来的设定时间执行
- 由定时器执行的函数称为定时器的 回调函数
- 定时器的周期: 从定时器启动到其回调函数执行之间的时间

当定时器的周期到期时，定时器的回调函数会被执行

- 在使用软件定时器之前，必须明确地创建它
- 定时器到期时间是相对于发送命令的时间计算的， 而不是相对于处理命令的时间计算的
- 一次性定时器 && 自动重载定时器



- 我们可以手动重置软件定时器



事件组

- 事件位：标志任务是否发生
- 事件组 = 一堆事件位

内存管理

- 每次创建任务、队列、互斥锁、软件定时器、信号量或事件组时，RTOS 内核都需要 RAM
- RAM 可以 从 RTOS API 对象创建函数内的 RTOS 堆自动动态分配， 或者可以由应用程序编写者提供
    - 由此可见，freeRTOS本身对于内存的管理还是很欠缺的，所以如果用RUST重构，会更安全一些

FatFS工作原理

FATFS 系统结构

FatFs 通过分层设计实现软硬件接口：

1. **应用层**：

    发者直接调用的文件操作接口，提供了一套完整的、与平台无关的 API，用于管理文件和目录。提供标准文件操作 API。应用层通过与文件系统层的 FIL、DIR 等对象封装文件操作，无需直接操作硬件。

2. **文件系统层**：

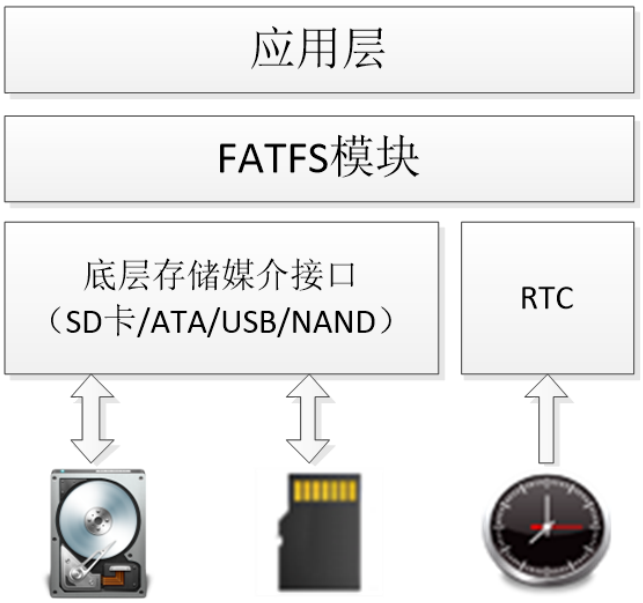
    负责解析 FAT 文件系统结构、管理文件和目录逻辑，并与硬件抽象层协作完成数据存取。实现了 FAT 文件读／写协议。提供了 `ff.c` 和 `ff.h` 。除非有必要，使用者一般不用修改，使用时将头文件直接包含进去即可。

    - FATFS 结构体：描述一个逻辑磁盘的文件系统状态，每个挂载的磁盘对应一个实例。
    - FIL 结构体：表示一个打开的文件，跟踪文件读写状态。

- DIR 结构体：用于目录遍历，记录当前目录位置。

3. **硬件抽象层**：

需用户实现的底层驱动接口（diskio.c）。包括存储媒介读／写接口（disk I / O）和供给文件创建修改时间的实时时钟，需要我们根据平台和存储介质编写移植代码。



Rust 的优越性(与C++相比)

与Rust相同，C++也是一种高级通用编译语言，其代码通常用于需要高速、并发性的程序，同时也以帮助开发者真正接近操作硬件而闻名。

首先，C++拥有更大的社区、更广泛的用例和更多框架；而 Rust 在安全性、速度以及防止由于静态类型特性而导致的错误、不安全代码方面表现更好。下面将从技术的角度对两种语言进行比较。

内存安全

因为垃圾收集等功能会降低性能，大多数系统级语言不提供自动内存管理。为了保持速度，C++牺牲了内存安全性，这是一个显著的缺点。用户必须手动管理内存，在原本 FreeRTos 的实现中，任务完成后需要专门调用函数确保内存释放，这对系统的性能和安全性带来了挑战。

为了解决这个问题，Rust 使用了所有权系统，它全面强制并提高了其内存安全性，基本上消除了对任何手动内存管理程序的需要。

编译

C++ 和 Rust 的完整构建时间大致相同，具体取决于 C++ 项目正在实现的模板数量（模板越多，速度越慢）。C++ 通常在增量编译方面做得更好，而 Rust 的编译器以友好而著称。它提供了有用的错误消息和一流的工具。

使用

大多数同时使用 Rust 和 C++ 的人都说 Rust 更易于使用，因为它具有明确定义的语义以及防止不需要/未定义行为的能力。同样，C++ 功能繁多，很难跟踪。

总而言之，Rust更适合于内存安全和并发，适合于编写实时操作系统，能够提高其运行速度和可靠性，避免出现C / C++导致的内存错误。

FreeRTOS可拓展模块及其应用场景

背景简介

FreeRTOS作为全球使用最广泛的实时操作系统（**RTOS**）之一，其代码结构清晰（仅3个核心文件），适合学生深入理解**任务调度、内存管理、中断机制**等操作系统核心原理。同时，FreeRTOS的模块化设计允许开发者按需扩展，但社区贡献多集中在基础功能，**针对特定场景的优化模块仍有空白**

已有的FreeRTOS库

FreeRTOS Plus 库

- **核心功能**：官方提供的增强模块合集，包含以下子模块：
  - **FreeRTOS+TCP**：轻量级 TCP/IP 协议栈，支持 IPv4/IPv6，适用于需要网络连接的嵌入式设备（如工业网关、智能家居控制器）。
  - **FreeRTOS+FAT**：嵌入式文件系统，支持 SD 卡和 RAM 磁盘，用于数据存储密集型场景（如数据采集设备）。
  - **FreeRTOS+CLI**：命令行接口工具，便于设备调试和远程控制。
- **应用场景**：物联网边缘计算、工业自动化控制、低功耗传感器网络。

FreeRTOS Core 库

`RTOS` 基础功能实现，包括任务调度器（支持协作/抢占式）、队列、信号量、软件定时器、事件组、内存管理等。作为所有 `FreeRTOS` 衍生库的基石，长期保持稳定更新，未来重点优化多核支持和 `RISC-V` 架构适配。

适用于 AWS IoT 的 FreeRTOS 库

- **核心功能**：与 Amazon Web Services 深度集成，提供安全连接、OTA 更新、设备管理等功能。
  - **安全启动**：基于硬件加密的固件验证机制。
  - **MQTT 协议支持**：实现与 AWS IoT Core 的高效通信。
- **应用场景**：智能农业监控系统、远程医疗设备、车联网终端。

FreeRTOS Lab 库

- **核心功能**：实验性功能模块，由社区贡献但尚未正式发布，例如：
  - **FreeRTOS-Kernel-Nano**：超轻量级内核，适用于 RAM < 4KB 的微控制器。
  - **FreeRTOS-SMP**：对称多核任务调度原型。
- **应用场景**：超低资源设备（如可穿戴设备）、多核处理器原型开发。

可扩展方向及其应用场景

动态任务优先级调节器

- **背景**：传统 FreeRTOS 任务优先级静态分配难以适应实时性动态变化场景。
- **功能设计**：
  - 基于任务执行历史（如响应时间、资源占用率）动态调整优先级。
  - 集成机器学习模型预测任务负载（需轻量化算法支持）。
- **应用场景**：
  - 无人机飞控系统（实时任务优先级随环境变化调整）。
  - 智能机器人多任务协作（紧急避障任务自动提升优先级）。

轻量级资源监控模块

- **背景**：嵌入式设备资源有限，需实时监控防止内存泄漏或 CPU 过载。
- **功能设计**：
  - 提供任务内存占用、堆栈使用率、CPU 负载的实时统计接口。
  - 支持阈值告警与自动触发安全模式（如任务挂起、系统重启）。
- **应用场景**：
  - 医疗设备（如心电图监测仪）的资源异常检测。
  - 工业 PLC 控制系统的故障预判。

动态内存分配优化器

- **背景**：FreeRTOS 默认内存分配策略（如 heap_4.c）易产生碎片，影响长期运行稳定性。
- **功能设计**：
  - 结合 TLSF（Two-Level Segregate Fit）算法实现低碎片化内存管理。
  - 支持按应用场景切换分配策略（如实时任务优先使用块内存池）。
- **应用场景**：
  - 长期运行的智能电表（减少内存碎片导致的系统崩溃）。
  - 高并发通信网关（优化多线程内存分配效率）。

低功耗任务调度优化器

- **背景**：物联网设备常受限于电池续航，需在保障实时性的前提下最大化能效。
- **功能设计**：
  - 基于任务执行周期动态调整 CPU 频率（如空闲时切换至低功耗模式）。

    - 支持外设智能休眠（如关闭未使用的传感器或通信模块）。
- **应用场景**：
    - 环境监测传感器节点（如温湿度传感器，延长野外部署寿命）。
    - 可穿戴健康设备（如智能手表的间歇性数据采集）。

FatFS可扩展方向

由于 FatFS 设计初衷是轻量级和简单，因此缺乏内置的安全功能（如加密和访问控制）是合理的，但考虑到一些场景下的安全相关需求，可以考虑以增加性能开销和复杂性为代价，增添文件加密与访问控制模块。

此外，FatFS可以拓展日志记录与事务处理功能，便于记录历史行为以及增加操作的安全性。

最后，可以考虑增加快照功能，以便在对系统进行了损害之后能够复原。

文件加密

FatFS 作为一个轻量级的 FAT 文件系统模块，专注于嵌入式系统的基本文件操作功能，而不是安全特性，因此现有的FatFS无内置加密功能。

ESP-IDF 框架中提到可以使用闪存加密与 FATFS 结合，但这是硬件级别的加密，而不是 FatFS 本身的功能。例如，在 Any way to enable encryption along with FATFS? (IDFGH-7251) · Issue #8842 · espressif/esp-idf 中，讨论了在 ESP32 上启用闪存加密，但明确指出 FatFS 本身不处理加密。此外，还有专利如 CN107111726A - The file encryption of FAT file system is supported - Google Patents 提到支持 FAT 文件系统的文件加密，但这不是 FatFS 的标准功能，而是外部实现的扩展。

可以使用对称加密算法如 AES-128 或 AES-256，因其性能较高，适合嵌入式系统。具体实现时，需要修改 `f_write` 函数与 `f_read` 函数，并存储加密密钥，可以使用使用硬件安全模块（如 TPM）或嵌入式安全元件（如 Secure Element）保护密钥，防止被提取。实现时，要注意到性能开销、兼容性等问题。

访问控制

通过查阅 FatFS 官方文档和配置选项（如 FatFs Module Application Note），没有发现内置的访问控制功能。FatFS 的线程安全配置（如 FF_FS_REENTRANT）允许通过互斥锁实现多任务并发访问控制，但这仅限于防止多个任务同时操作文件系统，而不是提供用户级别的读写权限管理。例如，在 FatFs Module Application Note 中，提到通过 ff_mutex_create/ff_mutex_delete/ff_mutex_take/ff_mutex_give 函数实现线程安全，但这仅用于同步访问，而不是控制哪个任务可以读写哪个文件。搜索“FatFS 访问控制”时，相关结果如 ASF 文档 ASF Source Code Documentation 提到“ctrl access modules”，但这指的是控制存储设备的接口，而不是文件级别的访问权限。

因此，可以确定现有的 FatFS 不包含访问控制模块。

我们需要实现基于用户或任务的权限表，记录每个文件的读、写、执行权限。可以基于 FreeRTOS 的任务 ID 实现，每个任务有自己的权限级别（如管理员、普通用户）。具体过程中，需要修改 `f_open` 函数，在打开文件前，检查任务是否有权限访问该文件；在文件系统中添加权限存储机制，可以在文件头存储权限元数据（如 4 字节表示读写权限）；权限表可以存储在 FatFS 的私有数据区或单独的文件中，可以使用 FAT 文件的扩展属性。

日志记录

日志记录通常指保持文件系统操作的记录，常见用于审计或调试。要实现 FatFS 的日志记录，可以创建一个专门的日志文件（如 "filesystem.log"），并为每个文件操作（如打开、写入、删除）编写包装函数。这些函数在调用原 FatFS 函数前后记录操作详情，包括时间戳、操作类型和文件路径。使用 FreeRTOS 时，确保通过互斥锁（mutex）保护日志文件访问，以保持线程安全。要注意的是，日志记录会增加性能开销，建议缓冲日志或仅记录关键操作以优化性能。

事务处理

事务处理在文件系统上下文中通常指执行一系列操作作为一个单元，要么全部成功要么全部失败（原子性），这对于维护数据完整性尤为重要。通过查阅 FatFS 官方文档和配置选项，没有发现内置的事务处理功能。FatFS 的线程安全配置（如 FF_FS_REENTRANT）允许通过互斥锁实现多任务并发访问控制，但这仅限于防止多个任务同时操作文件系统，而不是提供事务完整性。

事务处理需要确保一系列操作要么全部成功要么全部失败。可以在 FatFS 上实现事务管理器，通过创建临时文件执行操作，若成功则提交（例如移动文件到最终位置），若失败则回滚（删除临时文件）。为处理系统崩溃，可在启动时检查未完成事务并尝试完成或回滚，但这可能较为复杂。对于简单场景，临时文件方法已足够；复杂场景需记录事务日志以支持恢复。

快照

快照在文件系统上下文中通常是指在特定时间点创建文件系统的只读副本，用于备份、数据恢复或比较变化。常见的实现方式包括：

- **完整复制**：复制整个文件系统到单独区域。
- **复制写入（Copy-on-Write, COW）**：快照共享原始数据，修改时将新数据写入新位置，保持快照不变。
- **增量快照**：仅存储与前一快照的差异，节省空间。

可行性分析

重构可行性

C / C++ 与 Rust 的混合编译

Rust 和 C 代码之间的互操作性始终依赖于两种语言之间的数据转换。在嵌入式项目中包含 Rust 的一个常见要求是将 Cargo 与您现有的构建系统（例如 make 或 cmake）结合使用。

在 Rust 项目中使用 C 或 C++ 代码主要包括两个部分：

- 封装暴露的 C API 以便在 Rust 中使用
- 构建你的 C 或 C++ 代码，以便与 Rust 代码集成

由于 C++ 没有稳定的 ABI 供 Rust 编译器定位，因此建议在将 Rust 与 C 或 C++ 结合使用时使用 `C` ABI。

定义接口

在从 Rust 使用 C 或 C++ 代码之前，需要在 Rust 中定义链接代码中存在的数据类型和函数签名。在 C 或 C++ 中，你需要包含一个头文件（`.h` 或 `.hpp`）来定义这些数据。在 Rust 中，需要手动将这些定义转换为 Rust，或者使用工具生成这些定义。

构建 C / C++ 代码

由于 Rust 编译器不知道如何直接编译 C 或 C++ 代码（或来自任何其他语言的代码，这些语言提供 C 接口），因此需要提前编译你的非 Rust 代码。

对于嵌入式项目，这通常意味着将 C/C++ 代码编译为静态存档文件（例如 `cool-library.a`），然后在最终链接步骤中将其与你的 Rust 代码组合在一起。

如果你要使用的库已经作为静态存档文件分发，则无需重建代码。只需如上所述转换提供的接口头文件，并在编译/链接时包含静态存档文件即可。

如果你的代码以源项目的形式存在，则需要将你的 C/C++ 代码编译为静态库，方法是触发你现有的构建系统（例如 `make`、`CMake` 等），或者通过移植必要的编译步骤来使用名为 `cc` crate 的工具。对于这两个步骤，都需要使用 `build.rs` 脚本。

部署与测试

FreeRTos 支持多种硬件，考虑使用在 Windows / Linux 上的移植或者 QEMU 运行。因为 QEMU 的支持与社区较为完善，所以选择在 QEMU 上进行部署。

QEMU

QEMU 是一个通用的开源机器模拟器和虚拟机，允许用户连接 GDB 并调试系统软件映像

GDB (GNU Debugger)

GDB 是一种多功能的调试工具，允许开发人员在程序运行时或崩溃后检查程序的状态。对于嵌入式 Rust，GDB 通过 OpenOCD 或其他调试服务器连接到目标系统，以与嵌入式代码交互。GDB 是高度可配置的，支持远程调试、变量检查和条件断点等功能。它可以在各种平台上使用，并且广泛支持 Rust 特有的调试需求，例如漂亮的打印以及与 IDE 的集成。

拓展模块可行性

FreeRTOS与FatFS结合的可行性

存储设备驱动实现

- **硬件抽象层（HAL）**

  FatFS通过 `diskio.c` 的底层函数（如 `disk_read` 、 `disk_write` ）与物理存储设备交互。开发者需根据硬件平台（如STM32、ESP32）实现这些函数：

  - **SD卡**：需实现SDIO或SPI协议驱动，支持块读写（Block Size通常为512字节）。
  - **SPI Flash**：需实现Flash的扇区擦除（如4KB擦除）和读写函数，并适配FatFS的扇区映射逻辑。
  - **NAND Flash**：需处理坏块管理和ECC校验，可能需要额外中间层（如Flash转换层FTL）。

- **FreeRTOS任务调度与阻塞**
  存储设备的读写操作可能阻塞当前任务（如等待SPI传输完成）。此时FreeRTOS会自动切换任务，**需确保底层驱动在阻塞期间释放CPU。**

多存储介质支持

FatFS支持多个逻辑驱动器（如 `"0:/"` 、 `"1:/"` ），通过 `BYTE pdrv` 参数区分。在 FreeRTOS 中可为每个卷分配独立的互斥锁，避免并发冲突。

FreeRTOS更多模块拓展的可行性

技术可行性

- **架构开放性**
  - FreeRTOS 核心代码仅包含 3 个文件（ `tasks.c` , `queue.c` , `list.c` ），其余功能均以模块化形式实现（如 `FreeRTOS+TCP` 、 `FreeRTOS+FAT` ）。
  - 开发者可通过 Hook 函数、内核 API 和回调机制灵活扩展功能，无需修改内核代码。
- **资源占用控制**
  - 新模块可通过条件编译（ `#ifdef` ）实现按需加载，避免资源浪费。
  - 轻量化设计（如 TLSF 内存管理算法）可确保模块在资源受限设备中运行。
- **兼容性保障**
  - FreeRTOS 提供标准接口（如 `xTaskCreate()` 、 `xQueueSend()` ），新模块只需遵循接口规范即可与内核协同工作。
  - 支持多平台（ARM、RISC-V、Xtensa）和多种编译器（GCC、IAR、Keil），扩展模块可复用现有适配层。

社区与生态支持

- **官方支持**

- Amazon 主导的 FreeRTOS 长期维护计划（LTS）持续更新核心功能，并提供认证库（如 AWS IoT 设备 SDK）。
- FreeRTOS Lab 项目孵化实验性功能（如 SMP 多核支持），为社区扩展提供参考。

- **开发者社区贡献**
  - 开源社区已贡献大量实用模块（如 FreeRTOS-CPP 面向对象封装）。
  - 第三方厂商提供硬件适配驱动（如 ESP32 的 Wi-Fi/BLE 协议栈）。
- **行业合作案例**
  - **工业领域**：西门子将 FreeRTOS 用于 PLC 控制器，扩展了 Modbus 协议栈和冗余通信模块。
  - **消费电子**：乐鑫 ESP32 系列芯片内置 FreeRTOS，扩展了低功耗 Wi-Fi 管理功能。

FatFs更多模块拓展的可行性

丰富的社区资源

FatFS作为一种广泛应用于嵌入式系统的文件系统，拥有活跃的开发者社区和丰富的学习资源。例如，CSDN文库中有大量关于FatFS的教程和案例，包括如何在不同硬件平台上集成FatFS，以及如何实现文件加密和访问控制。此外，FatFS的官方GitHub讨论区也是开发者交流和解决问题的重要平台。这些资源为开发者提供了丰富的学习和参考材料，降低了开发难度。

技术可行性

从技术角度来看，FatFS的文件加密和访问控制是可行的。对于文件加密，可以采用对称加密算法（如AES），其在嵌入式系统中具有较高的性能和安全性。通过修改FatFS的文件读写函数（如`f_read`和`f_write`），可以在文件操作过程中实现加密和解密。对于访问控制，可以通过扩展FatFS的功能，实现基于用户或任务的权限管理。虽然FatFS本身不支持访问控制，但开发者可以通过自定义实现来满足需求。此外，FatFS的开源性质使得开发者可以根据具体需求进行定制和优化，进一步增强了其技术可行性。

综上所述，FatFS在文件加密和访问控制方面具有较高的技术可行性和丰富的社区资源支持，能够满足嵌入式系统中对数据安全的需求。

创新点

拓展更多模块

FreeRTOS 作为轻量级实时操作系统，其模块化设计和开源特性为功能扩展提供了良好基础。同时，FreeRTOS的模块化设计允许开发者按需扩展，但社区贡献多集中在基础功能，**针对特定场景的优化模块仍有空白**

Rust与FreeRTOS的结合

- 目前嵌入式系统仍主要使用C/C++进行开发，本项目探索Rust与FreeRTOS的结合，使其在嵌入式领域具有更好的内存安全性和并发管理能力。
- 研究如何使用Rust来编写FreeRTOS任务、管理内存、进行任务间通信，为Rust在RTOS领域的应用提供参考方案。

C / C++ 与 Rust 的高效混合编译

- 研究如何在Rust项目中高效调用C/C++代码，探索稳定的ABI（应用二进制接口）设计方案，以便兼容现有的C/C++嵌入式代码库。
- 结合Cargo与CMake等构建工具，实现C/C++与Rust代码的无缝集成。

面向嵌入式系统的Rust优化

- 研究如何在资源受限的环境下使用Rust，包括如何减少Rust编译后生成的二进制体积、优化性能以及降低对系统资源的消耗。
- 通过QEMU等虚拟化工具进行部署与测试，提供一种高效的嵌入式开发调试方式。

FreeRTOS内核优化与任务调度研究

- 深入分析FreeRTOS的任务调度机制，结合Rust的特性优化任务间通信，提高系统运行效率。
- 研究不同的内存管理策略，并探索如何结合Rust的内存管理模型优化FreeRTOS的RAM使用。

项目价值

提升嵌入式系统的安全性与稳定性

- Rust的所有权系统和编译期内存安全检查能够有效避免C语言中的内存泄漏、空指针引用等问题，提高嵌入式系统的稳定性和安全性。
- 结合Rust与FreeRTOS的特性，可以构建更加健壮的实时嵌入式系统，降低因内存错误导致的系统崩溃风险。

优化嵌入式系统的性能

- Rust相比C/C++具有更好的并发管理能力，有助于提高多任务处理的效率。
- 通过对FreeRTOS的任务调度和内存管理策略的研究，可以优化任务间通信和调度策略，提高系统响应速度。

拓展 FreeRTOS 的功能

- 在 FreeRTOS 中集成 FATFS, 便可以支持文件的存储和管理功能。可以实现文件的创建、读取、写入、删除等操作，方便用户对数据进行持久化存储

- 拓展动态任务优先级调节器、轻量级资源监控模块、动态内存分配优化器，使其能够适应实时性动态变化场景、防止内存泄漏或 CPU 过载、避免 FreeRTOS 默认的内存分配策略（如 heap_4.c）产生碎片，影响长期运行的稳定性

参考文献

- 互操作性 - 嵌入式 Rust Book - Rust 编程语言
- The Embedded Rust Book
- FreeRTOS Github Repos
- 超详细的FreeRTOS移植全教程——基于srm32
- Freertos-CSDN,介绍,移植,启动,内存管理,任务调度,通信机制
- FreeRTOS入门基础（持续更新）
- mustrust-OSH2024
- FatFs Module Application Notes
- FatFs - Generic FAT Filesystem Module
- 文件系统详解（FatFS)
- 文件系统FATFS的移植教程
- STM32之HAL开发——FatFs文件系统移植
- FreeRTOS-Plus-FAT
- FreeRTOS Support Archive - use fatfs on freertos
- Stm32-FatFs-FreeRTOS - Github项目
- FreeRTOS LPC2148 演示（由 JC Wren 提供） 包括 FatFS 和 LPCUSB
- Rust 交叉编译-cross
- RISC-V Bytes: Rust Cross-Compilation
- rust 交叉编译，吐血整理
- 聊聊Rust与C语言交互的具体步骤
- Rust/C 语言混合编译
- The Cargo Book
- learningdaily.dev
- FreeRTOS 库 - FreeRTOS™
- Lab-Project-FreeRTOS-FAT
- FATFS在嵌入式操作系统FreeRTOS中的移植与应用-不知不识网

- rust-fatfs
- 【FatFS 文件系统全面精通】:从基础入门到高级操作的完整指南(7大技巧+5个案例+3个扩展) - CSDN文库
- https://github.com/Bsm-B/Stm32-FatFs-FreeRTOS