freeRTOS & FatFs结合以及重构的可行性探讨

Intro about RTOS

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。
RTOS 通常比通用操作系统体积更小、重量更轻，因此 RTOS 非常适用于内存、计算和功率受限的设备。

- RTOS 分类
  - **硬实时系统**：必须在绝对严格的时间限制内完成任务（如航空电子系统）
  - **软实时系统**：允许偶尔超时，但需保持整体实时性（如视频流处理）
- RTOS 关键特性
  - **抢占式调度**：高优先级任务可中断低优先级任务
  - **任务优先级**：支持 256 级优先级（FreeRTOS 支持 32 级）
  - **中断处理**：支持嵌套中断和中断服务例程（ISR）
  - **内存保护**：部分 RTOS 支持 MMU 或 MPU（如 FreeRTOS 的 MPU 支持）

调度

实时操作系统（RTOS）利用与通用（非实时）系统相同的原理来实现多任务处理，但两者的目标截然不同。这一差异主要体现在调度策略上。

实时嵌入式系统旨在对现实世界的事件作出及时响应。这些事件通常有截止时间，实时嵌入式系统必须在此之前响应，RTOS 调度策略必须确保遵守这些截止时间要求。

为在小型 RTOS（如 FreeRTOS）中实现这一目标，软件工程师必须为每个任务分配优先级。RTOS 的调度策略就是**确保能够执行的最高优先级任务获得处理时间**。

如果存在多个能够运行的同等最高优先级任务（既没有延迟也没有阻塞），则调度策略可以选择在这些任务之间“公平”地分配处理时间。

- 这种基本形式的实时调度并非万能，无法改变时间的快慢，应用程序编写者必须确保设定的时序约束在所选任务优先级安排下是可行的。

FreeRTOS

FreeRTOS 简介

FreeRTOS是指小型实时操作系统内核。作为一个轻量级的操作系统，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。

FreeRTOS 项目地址

- FreeRTOS的多任务处理与并发举例说明

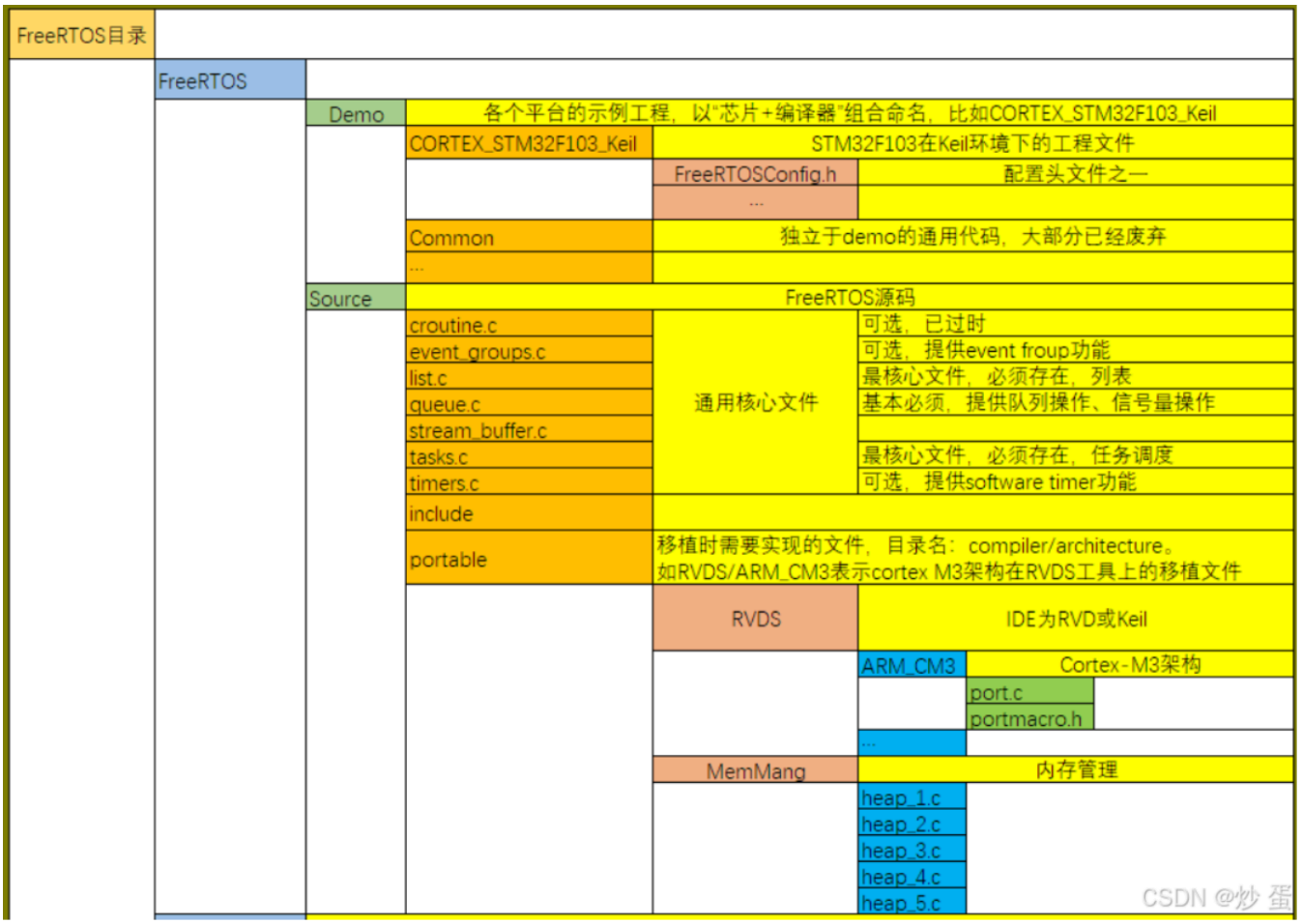
功能和特点（略）

- 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

FreeRTOS 核心功能详解

- **任务管理**：支持无限任务数，每个任务独立栈空间
- **内存管理**：提供 5 种内存分配策略（heap1-heap5）
- **同步机制**：支持互斥锁、信号量、事件组、队列
- **中断处理**：支持嵌套中断，提供专用 ISR API
- **调试工具**：支持可视化调试（如 FreeRTOS+Trace）

FreeRTOS 内核目录结构




```
FreeRTOS
    |
    +-Demo      Contains the demo application projects.
    |
    +-Source    Contains the real time kernel source code.


FreeRTOS
    |
    +-Source        The core FreeRTOS kernel files
        |
        +-include   The core FreeRTOS kernel header files
        |
        +-Portable  Processor specific code.
            |
            +-Compiler x    All the ports supported for compiler x
            +-Compiler y    All the ports supported for compiler y
            +-MemMang       The sample heap implementations
```

```
FreeRTOS
    |
    +-Demo
        |
        +-Common    The demo application files that are used by all
the demos.
        +-Dir x     The demo application build files for port x
        +-Dir y     The demo application build files for port y
```

- 核心 RTOS 代码包含在三个文件中，分别称为 task.c、queue.c 和 list.c
  它们位于 Source 目录中
- 同一目录包含两个 名为 timers.c 和 croutine.c 的可选文件，分别实现**软件计时器**和**协程**功能
- 每个受支持的处理器架构都需要少量的架构特定 RTOS 代码
  - 这是 RTOS 可移植层，位于 `FreeRTOS/Source/Portable/[compiler]/[architecture]` 子目录
  - 其中 `[compiler]` 和 `[architecture]` 分别是用于创建移植的编译器和移植运行的架构
- 出于 内存管理页面上所述的原因， 示例堆分配方案也位于可移植层中。各种示例 heap_x.c 文件位于 `FreeRTOS/Source/portable/MemMang` 目录中。

可移植层目录示例：

- 如果将 TriCore 1782 移植与 GCC 编译器一起使用： TriCore 特定文件 (port.c) 位于 FreeRTOS/Source/Portable/GCC/TriCore_1782 目录中。 除 FreeRTOS/Source/Portable/MemMang 外，所有其他 FreeRTOS/Source/Portable 子目录 都可以忽略或删除。
- 如果将 Renesas RX600 移植与 IAR 编译器一起使用： RX600 特定文件 (port.c) 位于 FreeRTOS/Source/Portable/IAR/RX600 目录中。除 FreeRTOS/Source/Portable/MemMang 外，所有其他 FreeRTOS/Source/Portable 子目录 都可以忽略或删除。
- 所有移植都是如此.....

两篇移植教程，可用做测试性能时的部署教程

超详细的FreeRTOS移植全教程——基于srm32

Freertos-CSDN博客

FreeRTOS 内核重构的可行性探讨

1. C 语言编写的代码潜在的问题是**内存安全**，借助 rust 语言的优秀机制（所有权，借用，生命周期等），我们可以在编译阶段就避免常见的内存安全问题（如空指针引用，悬空指针，内存泄漏和数据竞争等）
2. FreeRTOS 的内核并未**模块化**，结构并不够清晰，如 queue 和 tasks 两个 C 文件的代码行数为 2k 和 7k，但其中实现的功能其实可以详细的分门别类，形成清晰的结构
   - 如 queue 中开篇有大段的宏定义与数据结构的声明
   - 而后又是各个函数的具体实现
     - 函数又分为辅助函数和复杂函数
     - 由于需要考虑的条件过多，每一个函数的代码行都很长，再加上判断条件层层嵌套，全都放在一个 c 文件里对阅读和修改代码的人来说非常不友好

       而使用 rust 重构，我们不仅可以优化运行效率，也可以优化代码结构（因为 rust 使用包，crate 和模块可以管理不断增长的项目，其模块的优秀特性如模块内部可多层拆分以及使用 use 关键字将路径引入全局作用域等非常适合用来优化代码结构）

FatFS

> ## FATFS 简介
>
> FatFs是一个通用的文件系统(FAT/exFAT)模块，用于在小型嵌入式系统中实现FAT文件系统。 FatFs 组件的编写遵循ANSI C(C89)，完全分离于磁盘 I/O 层,因此不依赖于硬件平台。它可以嵌入到资源有限的微控制器中，不需要做任何修改。
>
> FATFS 项目地址
>
> ### 功能和特点
>
> - **文件操作**：支持打开 / 关闭 / 读写 / 删除文件
> - **目录操作**：支持创建 / 删除目录，遍历目录树
> - **文件属性**：支持读写时间戳、属性（只读、隐藏等）

> - **长文件名**：支持 Unicode 编码的长文件名（最长 255 字符）
> - **占用资源少**
>   对内存和代码空间的需求较低，非常适合资源受限的嵌入式系统。它可以根据具体应用需求进行裁剪，通过配置选项来控制功能的启用和禁用，从而减少系统资源的占用。例如，在一些小型的物联网设备中，FATFS 可以在有限的内存和存储资源下稳定运行
> - **支持多种 FAT 格式**
>   支持 FAT12、FAT16 和 FAT32 等多种 FAT 文件系统格式，具有广泛的兼容性。这使得它可以与各种使用 FAT 文件系统的设备和计算机进行数据交互，方便数据的共享和传输
> - **简单易用**
>   提供了简洁的 API 接口，开发者可以方便地进行文件和目录的操作，如创建、读取、写入、删除文件，以及创建、删除目录等。这些 API 函数与标准 C 库的文件操作函数类似，降低了开发者的学习成本

- FatFS 的核心文件其实只有两个，分别为 ff，和 diskio，其中 diskio 是**未实现**的部分，需要用户根据自己的硬件设备设计软件和硬件的数据传输接口
  - 这两个核心文件代码量也很多，有超过 9k+的代码量，因此可以选择重构，不过可能难度比较大，我更倾向于**使用 C 实现 diskio 的部分**然后**和 freeRTOS**结合

需要注意的事项

- 处理任务同步和互斥
  由于 FatFS 不是线程安全的，在多任务环境中需要使用 FreeRTOS 的同步机制（如互斥锁）来确保同一时间只有一个任务可以访问文件系统
  Eg.

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "ff.h"

#define TASK_STACK_SIZE 1024
#define TASK_PRIORITY 2

FATFS fs;
FIL file;
```

```c
SemaphoreHandle_t xFileMutex;

void file_management_task(void *pvParameters)
{
    FRESULT res;
    char buffer[] = "Hello, FatFS!";

    // 获取互斥锁
    xSemaphoreTake(xFileMutex, portMAX_DELAY);

    // 挂载文件系统
    res = f_mount(&fs, "", 1);
    if (res != FR_OK) {
        // 处理挂载失败的情况
        xSemaphoreGive(xFileMutex);
        while (1);
    }

    // 打开文件
    res = f_open(&file, "test.txt", FA_CREATE_ALWAYS | FA_WRITE);
    if (res == FR_OK) {
        // 写入数据
        UINT bytes_written;
        res = f_write(&file, buffer, sizeof(buffer), &bytes_written);
        if (res == FR_OK) {
            // 写入成功
        }
        // 关闭文件
        f_close(&file);
    }

    // 卸载文件系统
    f_mount(NULL, "", 1);

    // 释放互斥锁
    xSemaphoreGive(xFileMutex);

    // 任务结束
    vTaskDelete(NULL);
}

int main(void)
{
    // 创建互斥锁
```

```
    xFileMutex = xSemaphoreCreateMutex();

    // 创建文件管理任务
    xTaskCreate(file_management_task, "FileTask", TASK_STACK_SIZE,
NULL, TASK_PRIORITY, NULL);

    // 启动FreeRTOS调度器
    vTaskStartScheduler();

    // 不会执行到这里
    while (1);
}
```

- 与上述问题有关的FatFS 配置关键参数：

```
#define _FS_LOCK    2      // 支持文件锁定
#define _USE_MKFS    1      // 启用格式化功能
#define _WORD_ACCESS 1      // 强制字对齐（防止DMA错误）
```

Rust 与 C 的交叉编译

- 这一段没有了解太多，如果确定了做这个选题则继续深入了解，并且最可行的方式是联系往年的学长学姐向他们询问，这样可以更快的上手交叉编译
- 由于 FreeRTOS 和 FatFS 的 C 语言代码很长，并且使用了大量宏，这让重构变得困难了一点，尤其是部分重构
- 找到了几篇参考文章，让 AI 深度思考了一下总结如下：
- **Rust 与 C 语言交互**通过三种场景实现：**静态编译 C 代码**（使用`cc`库生成静态库）、**动态链接库绑定**（通过`bindgen`自动生成 Rust 绑定）、**复杂 C 库封装**（如 secp256k1，需手动编译依赖）。关键步骤包括**项目配置**（`Cargo.toml`添加`cc`和`bindgen`依赖）、**代码集成**（`extern "C"`声明或自动绑定）和**编译调试**（解决链接错误如`-lsecp256k1`缺失）。最终实现跨语言函数调用，例如通过`add(2,18)`输出 20，并通过测试验证功能。

3 种交叉编译步骤思维导图

```
Rust与C交互步骤
├─ 项目初始化
│  └─ cargo new --bin/--lib project
├─ 场景一：静态编译C代码
│  ├─ 配置Cargo.toml（cc依赖）
```

```
│   ├─ build.rs编译C文件为静态库
│   └─ main.rs通过extern "C"调用
├─ 场景二：动态链接库绑定
│   ├─ 添加bindgen依赖
│   ├─ 生成wrapper.h头文件
│   ├─ build.rs生成动态库+绑定代码
│   └─ main.rs引入自动绑定
├─ 场景三：封装复杂C库（如secp256k1）
│   ├─ 子模块引入C库
│   ├─ 手动编译C库（autogen.sh/configure/make）
│   └─ 处理链接错误（安装库文件）
└─ 关键工具
    ├─ cc：C代码编译
    └─ bindgen：自动生成Rust绑定
```

项目结构与配置

文件路径	作用	关键配置示例
`Cargo.toml`	依赖声明	`build-dependencies = ["cc", "bindgen"]`
`build.rs`	编译入口	`cc::Build::new().file("sample.c")`
`wrapper.h`	头文件包装器	`#include "sample.h"`
`sample_bindings.rs`	自动生成的 Rust 绑定代码	通过 `bindgen` 生成

核心场景对比

场景	工具链	关键步骤	优缺点
静态编译	`cc`	编译 C 为静态库，`extern "C"` 调用	无性能损失，但需静态集成
动态链接	`bindgen`	生成动态库 + 自动绑定代码	灵活，但需处理运行时依赖
复杂库封装	`cc` + 手动编译	子模块引入 + 编译 C 库	支持大型 C 项目，但配置复杂

关键代码实现

- **静态编译调用：**

```
// main.rs
#[link(name = "sample")]
extern "C" { fn add(a: c_int, b: c_int) -> c_int; }
let r = unsafe { add(2, 18) }; // 输出20
```

- **动态链接绑定：**

```
// build.rs
cc::new().file("sample.c").shared_flag(true).compile("sample.so");
let bindings = bindgen::default().header("wrapper.h").generate();
```

难点

- Rust 的学习（据说学习曲线比较陡峭）
- 由于重构任务可能是部分重构，或完全重构的过程中，两种情况下均会出现多语言编程的问题，这涉及到交叉编译一类的问题，也需要我们去学习了解
- FreeRTOS 和 FatFS 的代码量比较可观（虽然相比于其他的项目可能这并不算多），想要理解这些代码并尝试重构需要较长的学习过程，首先就是查阅两者的文档弄清楚两者的结构，各种 function 的功能等，然后就是尝试先部署，起码做到成功部署了才能尝试去重构
- 由于 FatFS 的 diskio 部分需要自行实现，因此我们也需要掌握一定的硬件知识（不算很多，感觉可能类似于上学期模电实验做的 FPGA，只要有文档的话不算什么问题）

总结

- **可行性？**
  一定是可行的，而且重构相对于创造而言，肯定相对来说容易上手一些，毕竟是从前人的肩膀上望远
- **推荐度？**
  由于往年的学长学姐都有用 rust 重构 freeRTOS 的选题，因此我们有很多前辈可以咨询路线，寻求帮助，这一点是很好的. 不过正是因为历年都有人做，所以我们起码得做到和历年的成果一样，而且最好是做的更好，这一点可能是比较有挑战性的

参考资料

- FreeRTOS kernel开发人员文档
- 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