基于 TM4C1294 在 ucosii 系统下驱动 SD(SPI)和 FATFS 移植和使用的说明文档

姓名：　胡纪甚

导师：　任家豪

部门：　观测系统部

职务：　嵌入式软件

时间：　2024.10.18

目　录

引 言

本学习文档旨在为 ZTT 嵌入式网络开发提供一个系统化的指导，同时也是我这段时间学习的总结。目的是在 TM4C1294NCPDT 微控制器上成功实现 uC/OS-II 操作系统的 SD 卡驱动移植和 FATFS 文件系统的使用。通过详细的步骤和示例，文档将涵盖从硬件连接到软件移植的各个方面，确保能够理解和掌握相关技术。

文章的大致结构如下：如果只想实现 SD 卡驱动编写，建议直接看第三章内容，如果只想实现 FATF 移植，建议直接看第四章内容。如果只想实现 FATF 的使用，建议直接看第五章内容：

1. TM4C1294 微控制器：介绍这款 MCU 与 SD 卡驱动相关的硬件连接和网络接口
2. SD 卡驱动：详细步骤写明了 SD 驱动步骤和注意事项。
3. 移植 FATFS 文件系统：详细步骤写明了 FATFS 文件系统的移植步骤和注意事项。
4. FATFS 驱动：实现初始化，具体任务，卸载这三个函数和说明。
5. 常见问题与解决方案：列出在开发过程中可能遇到的问题及其解决方案，帮助开发者快速排查和解决问题。

第一章　TM4C1294 微控制器和相关外设

1.1 硬件解读

SD 卡是一种广泛使用的闪存存储设备，FATFS 是一个轻量级的文件系统库。通过 FATFS，开发者可以在 SD 卡上实现文件的创建、读取、写入和删除等操作，简化了与 SD 卡的交互。FATFS 提供了一套标准的 API，使得在不同的硬件平台上使用 SD 卡变得更加方便和一致，同时支持多种文件操作和目录管理功能。在 PCB 设计中，SD 部分如图 1 所示，MCU 的控制引脚如图 2 所示。

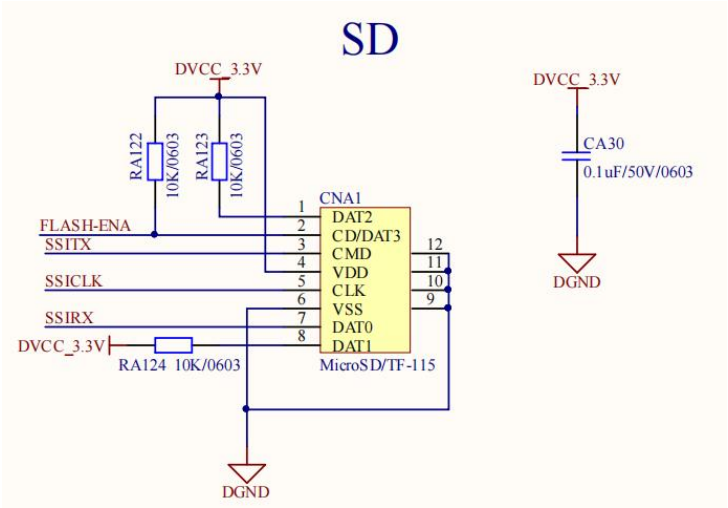

图 1 PCB 上的 SD

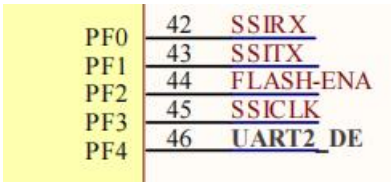

图 2 PCB 上 MCU 的 SD 驱动引脚

Ethernet 部分涉及到的组件和概念如下：

1. **CNA1**：连接器，用于连接 SD 卡。是一个 MicroSD/TF 卡槽，允许用户插入 MicroSD 卡。

2. **DVCC_3.3V**：这是 SD 卡接口的电源电压，提供 3.3 伏特的电压。

3. **CA30**：这是一个电容，通常用于滤除噪声或稳定电源线。

4. **FLASH-ENA**：使能信号，启动或禁用 SD 接口的闪存操作。

5. **SSITX、SSICLK、SSIRX**：用于与 SD 卡进行通信。SSITX 是发送线，SSICLK 是时钟线，SSIRX 是接收线。它们一起工作，允许设备通过 SPI（串行外设接口）协议与 SD 卡进行数据交换。

第二章　SD 卡驱动

SD 卡（Secure Digital Card）是一种广泛使用的闪存存储设备，常用于移动设备、数码相机、智能手机和嵌入式系统中。SD 卡具有高容量、低功耗和快速读写速度的特点，支持多种文件系统格式，其中 FAT（File Allocation Table）是最常用的格式之一。FAT 文件系统的简单性和广泛的兼容性使其成为 SD 卡的理想选择，尤其是在需要跨平台数据交换的场景中。

2.1 SD 卡驱动的硬件模块初始化

再次说到 SD 卡初始化，我们在初始化的时候，需要查阅两个文档。分别是《TM4C 开发手册》和《SD 卡协议标准》。分别对应硬件底层匹配和 SD 驱动初始化的相关步骤和任务。

SD 卡两种驱动方式有两种，分别是 SD 模式和 SPI 模式。此次初始化使用的是 SPI 模式。两个模式分别有各自的优缺点，其中 SPI 模式的优点是能够使用现成的主机，从而将设计的工作量减少到最小限度。缺点是 SPI 模式没有 SD 模式的性能好。

2.1.1 配置和初始化 SPI 硬件

1. SPI 协议概述

在初始化之前，我们必须要学会 SPI（串行外设接口）协议。SPI 协议的核心特性包括**全双工通信、主从模式、片选信号**控制以及灵活的**时钟速率设置**。为了有效地实现 SPI 通信，我们在代码中定义的四个函数必须根据这几个特性作为设计依据。

```
// 主从模式
int8_t Bsp_SpiInit(uint8_t spi_id);
// 全双工通信
uint8_t Bsp_RWByte(uint8_t spi_id, uint8_t txdata);
// 时钟速率设置
void Bsp_SpiSetSpeed(uint8_t spi_id, uint32_t ui32BitRate);
// 片选信号
void Bsp_SpiCS(uint8_t spi_id, uint8_t value);
```

2. 初始化 SPI 接口

这是第一个要实现的函数，也是这个工程最底层的基础，主要作用是初始化指定的 SPI 接口。SPI 协议要求在进行数据传输之前，必须配置相关的硬件参数。函数中的 spi_id 参数用于选择具体的 SPI 接口（SPI3），确保系统能够针对不同的 SPI 硬件进行初始化。函数首先启用 SPI 外设和相应的 GPIO 端口，这些 GPIO 端口包括时钟线（SCK）、主设备选

择线（SS）、数据接收线（MISO）和数据发送线（MOSI）。通过调用 GPIOPinConfigure()和 GPIOPinTypeSSI()函数配置了这些引脚的功能，使其能够正确地执行 SPI 通信。

在配置 SPI 参数时，SSIConfigSetExpClk()函数的参数包括时钟源、SPI 模式、工作模式、时钟频率和数据位数。SPI 模式（如 SSI_FRF_MOTO_MODE_0）定义了时钟极性和相位，确保发送和接收数据时的时序一致。而时钟极性和相位需要根据数据手册里的内容判断。如下图所示，默认低电平，所以时钟极性是 0，在第一个跳变沿数据采集，所以时钟相位是 0.所以选择模式 0。

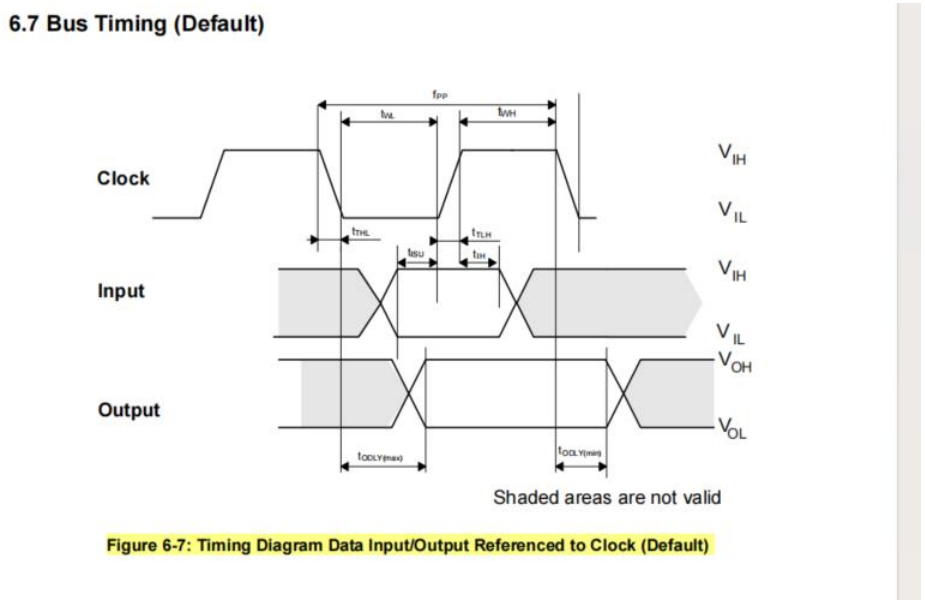

图 3 时钟极性和相位

工作模式（如 SSI_MODE_MASTER）指明该设备是主设备还是从设备，主设备负责生成时钟信号。时钟频率（1000000）决定了初始数据传输的速度，而数据位数（8）则指定了每次传输的数据量。最后，通过 SSIEnable 使能 SPI 接口，并清空接收缓冲区，确保在后续数据传输中不会受到旧数据的干扰。

```
int8_t Bsp_SpiInit(uint8_t spi_id)
{
    INT32U pui32DataRx;
    switch (spi_id) {
        case BSP_SPI3:
            SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI3);
            SysCtlPeripheralEnable(SPI3_SYS_PERIPH_GPIO);
            GPIOPinConfigure(SPI3_GPIO_CLK_CFG);
            GPIOPinConfigure(SPI3_GPIO_FSS_CFG);
            GPIOPinConfigure(SPI3_GPIO_RX_CFG);
            GPIOPinConfigure(SPI3_GPIO_TX_CFG);
```

```
            // 配置SPI 初始化的参数
            GPIOPinTypeSSI(SPI3_GPIO_BASE, SPI3_GPIO_CLK |  SPI3_GPIO_FSS |
                           SPI3_GPIO_RX | SPI3_GPIO_TX );
            SSIConfigSetExpClk(SSI3_BASE, BspGetSysClock(), SSI_FRF_MOTO_MODE_0,
                               SSI_MODE_MASTER, 1000000, 8);
            // 使能SPI 时钟总线
            SSIEnable(SSI3_BASE);
            while(SSIDataGetNonBlocking(SSI3_BASE, &pui32DataRx));  // 清除接收缓冲区
            break;
    }
    return 0;
}
```

3. 单字节收数据收发

SPI 是一种全双工通信协议，允许主机和从机同时发送和接收数据。每个 SPI 设备都有一个串行移位寄存器，主机通过向其寄存器写入数据来发起传输。在传输过程中，主机和从机通过 MOSI（主设备输出，从设备输入）和 MISO（主设备输入，从设备输出）信号线进行数据交换。

在 SPI 中，读和写操作是同步进行的。主机在进行写操作时，可以选择忽略接收到的数据；如果主机需要读取从机的数据，则必须先发送一个空字节以触发从机的响应。这意味着每次发送数据时，主机必然会接收到一个字节的数据。

由此特性可以得到以下函数，完全利用的全双工特性的单字节收发信息的底层函数。

```
uint8_t Bsp_RWByte(uint8_t spi_id, uint8_t data){
    uint32_t rec = 0;
    CPU_SR_ALLOC();
    OS_ENTER_CRITICAL();
    switch (spi_id) {
        case BSP_SPI3:
            while(SSIDataGetNonBlocking(SSI3_BASE, &rec));  // 清除接收缓冲区
            SSIDataPut(SSI3_BASE, data);
            while (SSIBusy(SSI3_BASE));
            SSIDataGet(SSI3_BASE, &rec);
            break;
    }
    OS_EXIT_CRITICAL();
    return (uint8_t)(rec & 0xff);
}
```

4. 片选信号与速率设置

在 SPI 通信中，片选信号（SS 或 CS）用于选择特定的从设备。当主设备需要与某个从设备进行通信时，它会将对应的片选信号拉低，表示该从设备被激活。每个从设备通常都有独立的片选线，这样主设备可以通过控制这些线来选择不同的从设备。片选信号的管理确保了在多设备环境中，只有一个从设备在任何时刻处于活动状态，从而避免数据冲突。

SPI 的速率选择是指主设备与从设备之间数据传输的速度。速率的选择需要考虑多个因素，包括设备的兼容性、信号完整性和应用需求。一般来说，主设备会根据从设备的最大支持速率来设置传输速率，以确保数据的可靠性和稳定性。过高的速率可能导致信号失真或数据丢失，因此在实际应用中，合理配置速率是确保 SPI 通信成功的关键。

```
void Bsp_SpiSetSpeed(uint8_t spi_id, uint32_t Speed){
    switch (spi_id) {
        case BSP_SPI3:
            SSIDisable(SSI3_BASE);
            SSIConfigSetExpClk(SSI3_BASE, BspGetSysClock(),
                               SSI_FRF_MOTO_MODE_0, SSI_MODE_MASTER,
                               Speed, 8);
            SSIEnable(SSI3_BASE);
            break;
    }
}
void Bsp_SpiCS(uint8_t spi_id, uint8_t value){
    switch (spi_id) {
        case BSP_SPI3:
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2,
                         value == 1 ? GPIO_PIN_2 : ~ GPIO_PIN_2);  // 低电平运行，高电平停止
            break;
    }
}
```

2.1.2 SD 驱动编写和底层基础功能配置

在完成了底层的 SPI 配置后，我们来到了这个人物的第一个关键节点，即 SD 驱动的编写和基本使用规则，在这里就要频繁的查阅《SD2.0 协议标准完整版》这个手册。

1. SD 卡初始化

SD 卡的初始化过程是确保卡与主机之间能够正确通信和操作的重要步骤。以下是根据您提供的 SD 2.0 协议标准完整版 PDF 文件中的内容，详细讲述 SD 卡初始化的全过程：

① 电源开启和卡检测（Power Up and Card Detection）

初始化过程开始于 SD 卡的电源开启。在这个阶段，主机（Host）需要确保卡（Card）接收到的电压在规定的最小电压（VDD_min）之内，并在 250 毫秒内供电至卡。同时，主机需要通过检测 CD/DAT3 引脚上的上拉电阻来识别卡是否已插入。

② 发送 CMD0 命令（Sending CMD0 Command）

一旦卡检测到电源，主机发送 CMD0（GO_IDLE_STATE）命令将所有卡重置为空闲状态（Idle State）。这个命令不要求卡做出响应，目的是将卡置于一个已知的起始状态。

③ 检查电压范围（Checking Voltage Range）

在 CMD0 命令之后，主机通过发送 CMD8（SEND_IF_COND）命令来检查卡是否支持当前提供的电压范围。这个命令包含一个参数，指定了主机提供的电压范围和校验模式。如果卡支持这个电压范围，它会返回一个响应，回显参数中的电压范围和校验模式。

④ 发送 ACMD41 命令（Sending ACMD41 Command）

在确认电压范围之后，主机发送 ACMD41（SD_SEND_OP_COND）命令来初始化卡。这个命令用于确定卡的操作系统条件，并且如果卡是高容量卡，还可以通过这个命令来识别。ACMD41 命令需要在 CMD8 命令之后发送，以确保卡已经准备好接受进一步的初始化命令。

⑤ 卡状态查询（Card State Inquiry）

在发送 ACMD41 命令之后，主机需要不断查询卡的状态，直到卡完成初始化。这是通过发送 ACMD41 命令并检查响应中的忙标志位（Busy bit）来完成的。如果忙标志位被设置，表示卡还在初始化过程中；如果忙标志位被清除，表示卡已经初始化完成。

⑥ 卡识别（Card Identification）

一旦卡完成初始化，主机发送 CMD2（ALL_SEND_CID）命令来请求所有卡发送它们的卡识别号码（CID）。这个命令会导致每个卡发送其唯一的识别信息。

⑦ 分配相对卡地址（Assigning Relative Card Address）

在卡发送了 CID 之后，主机发送 CMD3（SEND_RELATIVE_ADDR）命令来请求卡分配一个相对卡地址（RCA）。这个 RCA 是一个较短的地址，用于在数据传输模式中标识卡。

⑧ 选择卡（Selecting the Card）

最后，主机再次发送 CMD7（SELECT/DESELECT_CARD）命令，使用之前分配的 RCA 来选择卡。这个命令将卡从待机状态（Stand-by State）切换到传输状态（Transfer State），使其准备好进行数据传输。

完成以上步骤后，SD 卡初始化过程结束，卡已经准备好进行数据传输或其他操作。需要注意的是，这个过程中的每一步都是基于前一步的成功完成。如果任何一个步骤失败，都需要进行错误处理，可能包括重新初始化或停止操作。可以直接看图 3。

除了以上的内容，还有几个需要注意的地方，有些是细节，有些是 SD 卡自带的一些小 bug：

① 在初始化前，需要在低速模式下发送至少 74 个脉冲来激活 SD 卡的收发功能。

② 在初始化前，需要发送以下信号进行重置：

0x40, 0x0, 0x0, 0x0, 0x0, 0x95

③ 在 SPI 模式下，如果 sd 卡操作失败，需要在高速模式下发送高电平 8 个额外时钟。

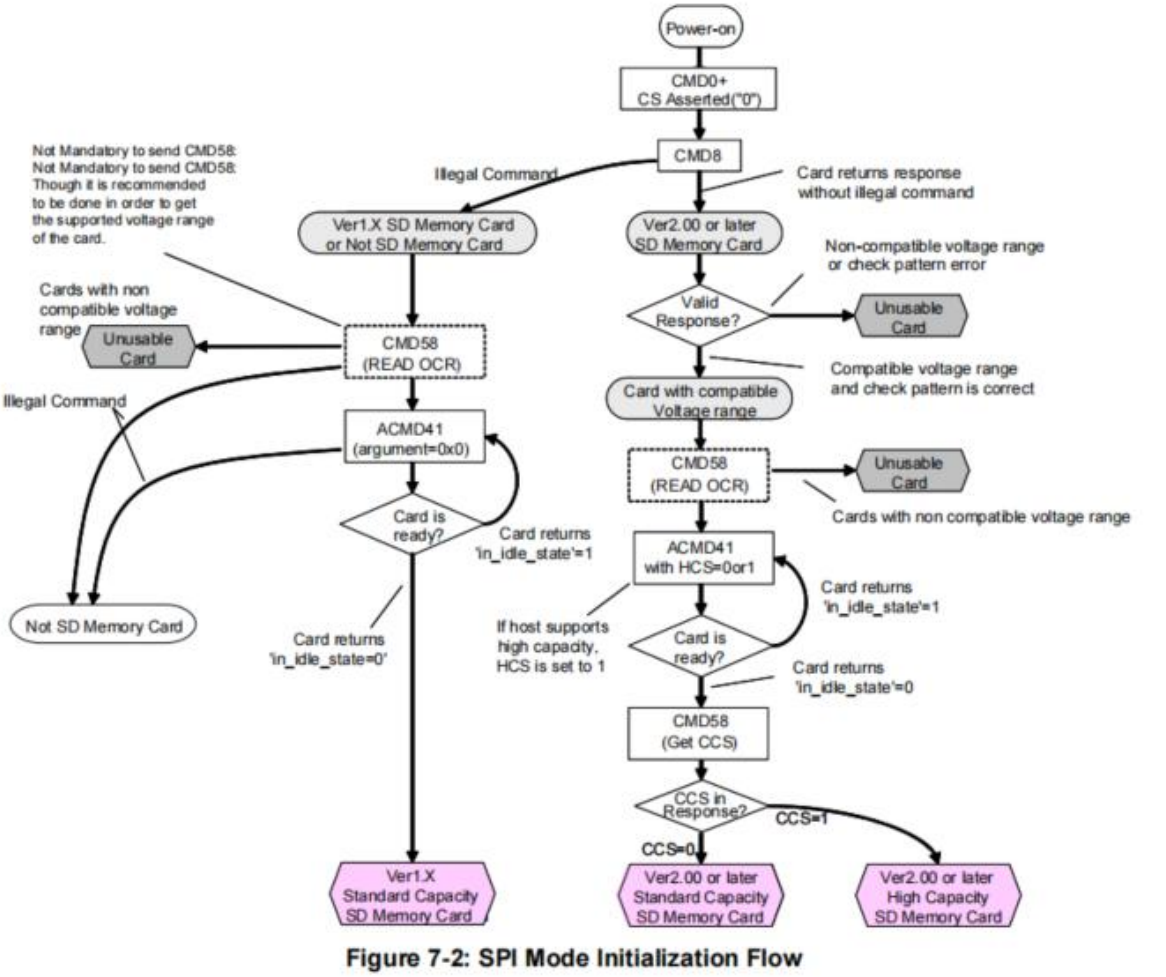

Figure 7-2: SPI Mode Initialization Flow

图 3 SD 卡驱动全流程

2. SD 卡功能模块

除了初始化，最重要的事情当然是功能了，由于使用的是 SPI 模式下的，所以根据 SPI 的特点编写了以下这些函数。

① 命令发送函数 (Drv_SdSendCmd)

命令发送函数的主要功能是向 SD 卡发送命令并接收其响应。该函数首先构建要发送的命令，包括命令字、参数和校验和。构建完成后，函数通过 SPI 接口将命令发送到 SD 卡，并在发送的同时读取 SD 卡返回的响应字节。这个响应通常是一个状态码，指示命令的执行结果。通过这种方式，函数能够有效地与 SD 卡进行通信，确保命令的正确执行。

② SPI 读写函数 (Drv_SdSpiRWByte)

SPI 读写函数负责通过 SPI 接口进行数据的读写操作。其逻辑是同步的：在发送数据的同时，函数会读取 SPI 接口返回的数据。这是 SPI 协议的特性，发送和接收是同时进行的。函数首先将要发送的数据字节通过 SPI 接口发送出去，然后立即接收并返回 SPI 接口返回的数据。这种设计确保了数据传输的高效性和可靠性。

③ 选择和取消选择函数 (Drv_SdSpiCs, Drv_SdDisselect)

选择和取消选择函数用于控制 SD 卡的选择信号（Chip Select, CS）。选择函数通过将 CS 引脚拉低来选择 SD 卡，使其准备好接收命令和数据。相反，取消选择函数则通过将

CS 引脚拉高来结束与 SD 卡的通信。这两个函数的设计确保了在与 SD 卡进行通信时，只有目标卡处于活动状态，从而避免了数据冲突。

④ SPI 速度设置函数 (Drv_SdSpiSetSpeed)

SPI 速度设置函数用于调整 SPI 通信的速度。该函数根据传入的速度参数配置 SPI 控制寄存器，以改变 SPI 的时钟频率。在初始化时，通常会将 SPI 设置为低速模式，以确保与 SD 卡的兼容性。完成初始化后，函数可以将 SPI 速度切换到高速模式，以提高数据传输效率。这种灵活的速度设置使得系统能够在不同的操作阶段优化性能。

⑤ 错误处理函数

在整个过程中，代码会根据不同的返回值进行错误处理。例如，在发送命令时，如果在规定的重试次数内没有收到有效响应，则认为初始化失败。此外，在识别 SD 卡类型时，如果出现错误，代码会将 sd_type 设置为错误类型，并在最后返回相应的错误码。这种错误处理机制确保了系统的健壮性，能够有效应对各种异常情况。

至此 SD 卡驱动完成！

第三章 移植 FATFS 文件系统

FATFS（File Allocation Table File System）是一种广泛使用的文件系统，特别适用于嵌入式系统和小型存储设备，如 SD 卡和 USB 闪存驱动器。FATFS 的设计基于 FAT 文件系统的原理，最初由微软在 1970 年代开发。其主要特点是简单、易于实现和高效，适合资源有限的环境。

FATFS 的核心结构是文件分配表（FAT），它用于跟踪存储设备上文件的存储位置。FAT 表实际上是一个数组，每个元素对应存储设备上的一个簇（cluster），记录该簇的状态（如空闲、已分配或坏簇）。当文件被创建或写入时，FATFS 会在 FAT 表中分配一个或多个簇，并将这些簇的链表结构记录下来，以便后续访问。

在 FATFS 中，文件和目录的管理是通过目录项（directory entry）实现的。每个文件或目录都有一个对应的目录项，包含文件名、文件大小、创建时间、修改时间等元数据。目录项的结构简单明了，使得文件的查找和管理变得高效。用户可以通过标准的 API 接口进行文件的创建、读取、写入和删除等操作，这些接口隐藏了底层的复杂性，使得开发者能够方便地进行文件操作。

FATFS 支持多种不同的 FAT 格式，包括 FAT12、FAT16 和 FAT32。FAT12 适用于小型存储设备，FAT16 则适用于中等容量的设备，而 FAT32 则能够支持更大的存储空间，适合现代大容量存储设备。每种格式都有其特定的簇大小和最大文件大小限制。

在嵌入式系统中，FATFS 的优势在于其轻量级和高效性。由于其简单的结构，FATFS 能够在资源有限的环境中快速启动和运行。

3.1 FATFS 文件系统架构

先看这张图，红框部分就是我们需要使用的 FATFS 的文件系统。可以看到，直接与我们底层相联系的的 diskio.c，这也是我们需要改动的部分，完成对底层的适配，而 ff.c 和 ffsystem.c 属于系统的核心架构，几乎不用怎么修改，只需要调整几个使能信号和参数即可。而 ffunicode.c 则是支持其他语言文件辅助文件，暂时没有用到。

ff.c 是 FATFS 的主要实现文件，负责文件系统的核心功能。它包含了文件和目录的管理、读写操作、文件分配表的处理等关键逻辑。具体来说，ff.c 实现了文件的打开、关闭、读取、写入和删除等基本操作。每个操作都通过一系列的 API 接口提供给用户，隐藏了底层的复杂性，使得开发者能够方便地进行文件操作。此外，ff.c 还处理了文件系统的初始化和挂载过程，确保在访问存储设备之前，文件系统能够正确识别和配置。

ffsystem.c 则主要负责文件系统的系统级管理功能。它实现了一些与文件系统状态和配置相关的操作，例如文件系统的挂载、卸载、格式化和检查等。该文件中的函数通常涉及对 FAT 表的管理和维护，确保文件系统的完整性和稳定性。ffsystem.c 还处理了与存储设备的交互，包括读取和写入 FAT 表、更新目录项等。这些功能对于确保文件系统的正常运行至关重要，尤其是在处理存储设备的错误和坏簇时。

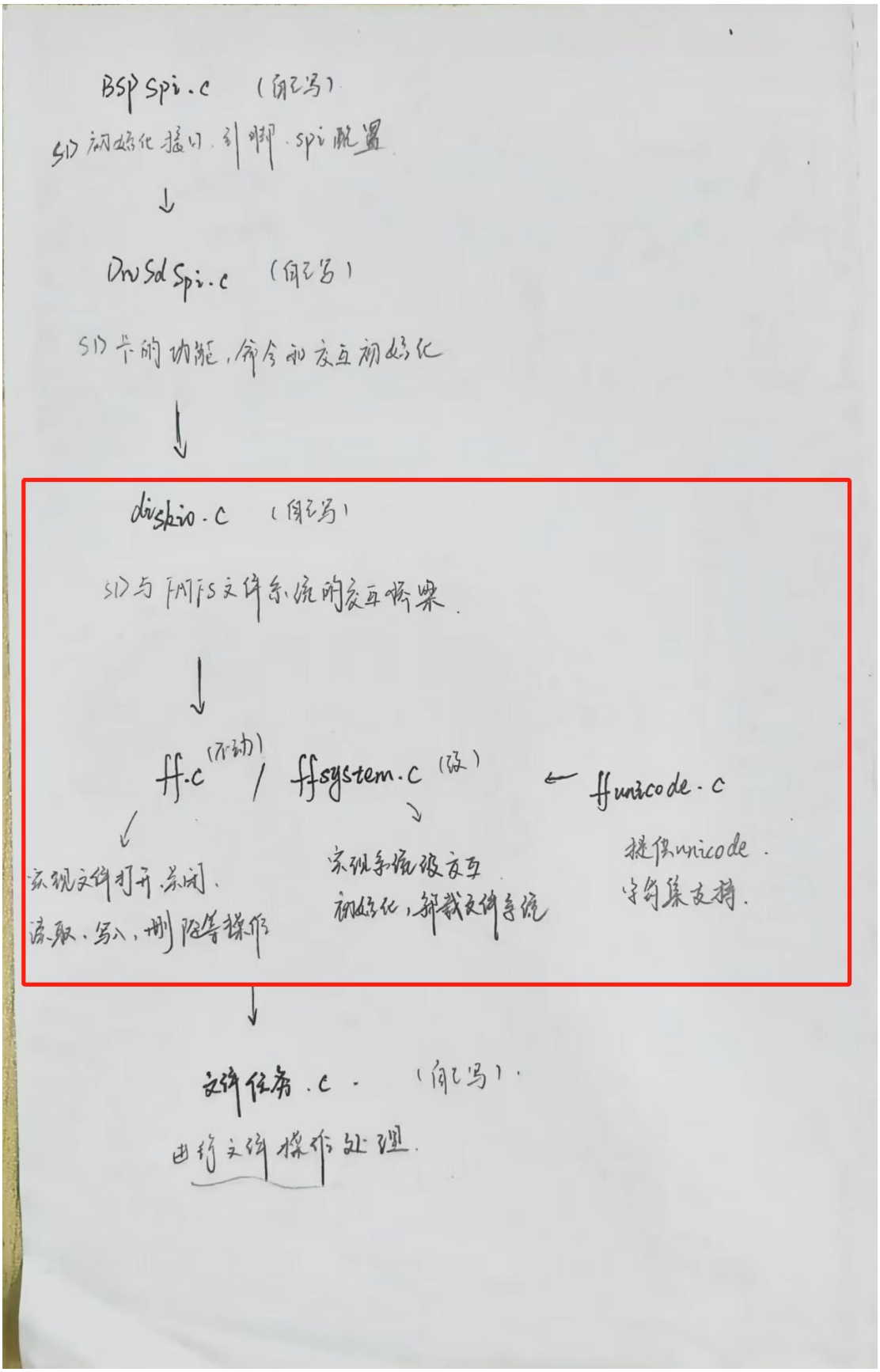

图 4 FATFS 文件系统架构示意图

3.2 diskio.c

这个文件实现了 FATFS 文件系统与物理存储设备（如 SD 卡）之间的接口。它定义了一系列函数，这些函数负责处理与 SD 卡的初始化、读写操作、状态检查和控制命令等相关的任务。通过这些函数，FATFS 能够在高层次上管理文件和目录，而不需要直接与硬件交互。

1. 状态检查和初始化

在 disk_status 和 disk_initialize 函数中，添加了对 SD 卡状态的检查和初始化逻辑。具体来说，disk_status 函数直接返回 RES_OK，因为 SD 卡在上电后是固定状态的。这种修改简化了状态检查的逻辑，确保在 SD 卡正常工作时能够快速返回状态。

2. 错误处理

在 disk_read 和 disk_write 函数中，增加了对参数的检查。例如，在 disk_read 中，如果 count 为 0，则返回 RES_PARERR，表示参数错误。这种修改提高了代码的健壮性，避免了在读取时出现无效的操作。

3. 读写操作的返回值处理

在 disk_read 和 disk_write 函数中，增加了对 SDReadDisk 和 SDWriteDisk 返回值的检查。如果返回值不为 0，则返回 RES_ERROR。这种修改确保了在读写操作失败时能够及时反馈错误，增强了系统的可靠性。

4. IO 控制函数的实现

在 disk_ioctl 函数中，增加了对不同控制命令的处理逻辑，如 CTRL_SYNC、GET_SECTOR_SIZE、GET_BLOCK_SIZE 和 GET_SECTOR_COUNT。这些修改使得代码能够更好地与 FATFS 的要求对接，提供必要的控制功能，确保文件系统能够正确获取和设置相关参数。

5. 时间获取函数的实现

在 get_fattime 函数中，增加了对 RTC（实时时钟）时间的获取逻辑。如果 RTC 时间有效，则返回相应的时间戳；否则，返回一个默认的时间。这种修改确保了即使在没有 RTC 支持的情况下，系统也能返回一个有效的时间戳，满足 FATFS 对文件时间戳的要求。

6. 宏定义的使用

代码中使用了#ifdef USE_DEV_SDHC 来控制 SD 卡相关的代码块。这种条件编译的方式使得代码更加灵活，能够根据不同的配置编译出适合特定硬件的版本。这种修改提高了代码的可移植性和适应性。

3.3 ff.c

先看看 ff.c 的部分头文件，这是我们之后进行文件操作所必须的：

```
1.  FRESULT f_open (FIL* fp, const TCHAR* path, BYTE mode);    /* Open or create a file */
2.  FRESULT f_close (FIL* fp);          /* Close an open file object */
3.  FRESULT f_read (FIL* fp, void* buff, UINT btr, UINT* br);   /* Read data from the file */
4.  FRESULT f_write (FIL* fp, const void* buff, UINT btw, UINT* bw); /* Write data to the file */
5.  FRESULT f_lseek (FIL* fp, FSIZE_t ofs);       /* Move file pointer of the file object */
```

```
6.  FRESULT f_truncate (FIL* fp);          /* Truncate the file */
7.  FRESULT f_sync (FIL* fp);          /* Flush cached data of the writing file */
8.  FRESULT f_opendir (DIR* dp, const TCHAR* path);      /* Open a directory */
9.  FRESULT f_closedir (DIR* dp);          /* Close an open directory */
10. FRESULT f_readdir (DIR* dp, FILINFO* fno);       /* Read a directory item */
11. FRESULT f_findfirst (DIR* dp, FILINFO* fno, const TCHAR* path, const TCHAR* pattern); /* Find firs
    t file */
12. FRESULT f_findnext (DIR* dp, FILINFO* fno);       /* Find next file */
13. FRESULT f_mkdir (const TCHAR* path);        /* Create a sub directory */
14. FRESULT f_unlink (const TCHAR* path);        /* Delete an existing file or directory */
15. FRESULT f_rename (const TCHAR* path_old, const TCHAR* path_new); /* Rename/Move a file or director
    y */
16. FRESULT f_stat (const TCHAR* path, FILINFO* fno);     /* Get file status */
17. FRESULT f_chmod (const TCHAR* path, BYTE attr, BYTE mask);   /* Change attribute of a file/dir */
18. FRESULT f_utime (const TCHAR* path, const FILINFO* fno);   /* Change timestamp of a file/dir */
19. FRESULT f_chdir (const TCHAR* path);        /* Change current directory */
20. FRESULT f_chdrive (const TCHAR* path);        /* Change current drive */
21. FRESULT f_getcwd (TCHAR* buff, UINT len);       /* Get current directory */
22. FRESULT f_getfree (const TCHAR* path, DWORD* nclst, FATFS** fatfs); /* Get number of free clusters
     on the drive */
23. FRESULT f_getlabel (const TCHAR* path, TCHAR* label, DWORD* vsn); /* Get volume label */
24. FRESULT f_setlabel (const TCHAR* label);       /* Set volume label */
25. FRESULT f_forward (FIL* fp, UINT(*func)(const BYTE*,UINT), UINT btf, UINT* bf); /* Forward data to
     the stream */
26. FRESULT f_expand (FIL* fp, FSIZE_t fsz, BYTE opt);     /* Allocate a contiguous block to the file
    */
27. FRESULT f_mount (FATFS* fs, const TCHAR* path, BYTE opt);   /* Mount/Unmount a logical drive */
28. FRESULT f_mkfs (const TCHAR* path, const MKFS_PARM* opt, void* work, UINT len); /* Create a FAT vo
    lume */
29. FRESULT f_fdisk (BYTE pdrv, const LBA_t ptbl[], void* work);  /* Divide a physical drive into some
     partitions */
30. FRESULT f_setcp (WORD cp);          /* Set current code page */
31. int f_putc (TCHAR c, FIL* fp);          /* Put a character to the file */
32. int f_puts (const TCHAR* str, FIL* cp);        /* Put a string to the file */
33. int f_printf (FIL* fp, const TCHAR* str, ...);      /* Put a formatted string to the file */
34. TCHAR* f_gets (TCHAR* buff, int len, FIL* fp);      /* Get a string from the file */
```

1. 文件操作类型函数

f_open 函数用于打开或创建一个文件。它的参数包括一个指向 FIL 结构的指针 fp，用于存储文件对象的信息；一个指向字符串的指针 path，表示文件的路径；以及一个字节 mode，指定打开文件的模式（如只读、写入或追加）。成功打开文件后，fp 将包含文件的状态和指针信息，供后续操作使用。

f_close 函数用于关闭一个已打开的文件。它的参数是一个指向 FIL 结构的指针 fp，表示要关闭的文件对象。关闭文件后，系统会释放与该文件相关的资源，确保数据完整性。

f_read 函数用于从文件中读取数据。它的参数包括一个指向 FIL 结构的指针 fp，一个指向缓冲区的指针 buff，表示要存储读取数据的地方，btr 表示要读取的字节数，br 是一个指向 UINT 的指针，用于返回实际读取的字节数。通过这个函数，用户可以从文件中获取所需的数据。

f_write 函数则用于向文件写入数据。它的参数与 f_read 类似，包括一个指向 FIL 结构的指针 fp，一个指向要写入数据的缓冲区的指针 buff，btw 表示要写入的字节数，bw 是一个指向 UINT 的指针，用于返回实际写入的字节数。这个函数使得用户能够将数据保存到文件中。

f_lseek 函数用于移动文件指针。它的参数是一个指向 FIL 结构的指针 fp 和一个 FSIZE_t 类型的偏移量 ofs，表示新的文件指针位置。通过调整文件指针，用户可以在文件中随机访问数据。

f_truncate 函数用于截断文件。它的参数是一个指向 FIL 结构的指针 fp，表示要截断的文件对象。截断操作会将文件的大小调整为当前文件指针的位置，删除指针后面的数据。

f_sync 函数用于刷新写入文件的缓存数据。它的参数是一个指向 FIL 结构的指针 fp，表示要刷新的文件对象。通过调用此函数，用户可以确保所有未写入的数据都被写入到存储介质中。

2. 目录操作类型函数

f_opendir 函数用于打开一个目录。它的参数包括一个指向 DIR 结构的指针 dp，用于存储目录对象的信息，以及一个指向字符串的指针 path，表示目录的路径。成功打开目录后，dp 将包含目录的状态和指针信息。

f_closedir 函数用于关闭一个已打开的目录。它的参数是一个指向 DIR 结构的指针 dp，表示要关闭的目录对象。关闭目录后，系统会释放与该目录相关的资源。

f_readdir 函数用于读取目录项。它的参数包括一个指向 DIR 结构的指针 dp 和一个指向 FILINFO 结构的指针 fno，用于存储读取到的目录项信息。通过这个函数，用户可以遍历目录中的文件和子目录。

f_findfirst 和 f_findnext 函数用于查找文件。f_findfirst 的参数包括一个指向 DIR 结构的指针 dp，一个指向 FILINFO 结构的指针 fno，一个指向字符串的指针 path，表示查找的路径，以及一个指向字符串的指针 pattern，表示查找的模式（如文件扩展名）。f_findnext 函数则用于查找下一个文件。通过这两个函数，用户可以在指定目录中查找符合条件的文件。

f_mkdir 函数用于创建子目录。它的参数是一个指向字符串的指针 path，表示要创建的目录路径。成功创建目录后，用户可以在该目录下进行文件操作。

f_unlink 函数用于删除文件或目录。它的参数是一个指向字符串的指针 path，表示要删除的文件或目录的路径。通过此函数，用户可以管理文件系统中的内容。

f_rename 函数用于重命名或移动文件或目录。它的参数包括两个指向字符串的指针，path_old 表示旧路径，path_new 表示新路径。通过此函数，用户可以方便地更改文件或目录的名称或位置。

3. 文件状态和属性操作

f_stat 函数用于获取文件状态。它的参数包括一个指向字符串的指针 path，表示要获取状态的文件路径，以及一个指向 FILINFO 结构的指针 fno，用于存储文件状态信息。通过此函数，用户可以获取文件的大小、创建时间等信息。

f_chmod 函数用于更改文件或目录的属性。它的参数包括一个指向字符串的指针 path，表示要更改属性的文件或目录路径，attr 表示新的属性值，mask 表示要更改的属性位掩码。通过此函数，用户可以控制文件的可读、可写和可执行权限。

f_utime 函数用于更改文件或目录的时间戳。它的参数包括一个指向字符串的指针 path，表示要更改时间戳的文件或目录路径，以及一个指向 FILINFO 结构的指针 fno，用于提供新的时间戳信息。通过此函数，用户可以更新文件的创建和修改时间。

4. 当前目录和驱动管理

f_chdir 函数用于更改当前目录。它的参数是一个指向字符串的指针 path，表示新的当前目录路径。通过此函数，用户可以在文件系统中导航到不同的目录。

f_chdrive 函数用于更改当前驱动。它的参数是一个指向字符串的指针 path，表示新的当前驱动路径。通过此函数，用户可以在多个逻辑驱动之间切换。

f_getcwd 函数用于获取当前目录。它的参数包括一个指向字符数组的指针 buff，用于存储当前目录路径，以及一个 UINT 类型的 len，表示缓冲区的长度。通过此函数，用户可以获取当前工作目录的信息。

5. 驱动和文件系统管理

f_getfree 函数用于获取驱动上的空闲簇数。它的参数包括一个指向字符串的指针 path，表示要查询的驱动路径，以及两个指针 nclst 和 fatfs，分别用于返回空闲簇的数量和指向文件系统对象的指针。通过此函数，用户可以了解驱动的存储状态。

f_getlabel 和 f_setlabel 函数分别用于获取和设置卷标。f_getlabel 的参数包括一个指向字符串的指针 path，用于指定驱动路径，一个指向字符数组的指针 label，用于存储卷标，以及一个指向 DWORD 的指针 vsn，用于返回卷序列号。f_setlabel 的参数是一个指向字符数组的指针 label，表示要设置的新卷标。通过这两个函数，用户可以管理驱动的标识信息。

6. 数据流和文件系统管理

f_forward 函数用于将数据转发到流。它的参数包括一个指向 FIL 结构的指针 fp，表示目标文件，func 是一个指向函数的指针，用于处理数据，btf 表示要转发的字节数，bf 是一个指向 UINT 的指针，用于返回实际转发的字节数。通过此函数，用户可以将数据从一个源转发到文件。

f_expand 函数用于为文件分配连续的块。它的参数包括一个指向 FIL 结构的指针 fp，表示目标文件，fsz 表示要分配的大小，opt 表示分配选项。通过此函数，用户可以管理文件的存储空间。

f_mount 和 f_mkfs 函数用于挂载和创建 FAT 卷。f_mount 的参数包括一个指向 FATFS 结构的指针 fs,表示文件系统对象,path 表示要挂载的驱动路径,opt 表示挂载选项。f_mkfs 的参数包括一个指向字符串的指针 path，表示要创建的驱动路径，一个指向 MKFS_PARM 结构的指针 opt，用于指定创建选项，一个指向工作缓冲区的指针 work，以及一个 UINT 类型的 len，表示缓冲区的长度。通过这两个函数，用户可以管理文件系统的结构和状态。

6. 字符串和字符操作

f_putc、f_puts、f_printf 和 f_gets 函数用于处理字符和字符串的输入输出。f_putc 用于将一个字符写入文件，f_puts 用于将一个字符串写入文件，f_printf 用于格式化输出字符串到文件，f_gets 用于从文件中读取字符串。通过这些函数，用户可以方便地处理文本数据。

第四章 FATFS 驱动

4.1 FATFS 初始化函数

函数代码如下：

```
// 初始化，挂载文件系统void InitFileSystem() {

    // 尝试挂载文件系统
    res = f_mount(&fs, "", 1);
    if (res != FR_OK) {
        // 如果挂载失败，尝试创建文件系统
        MKFS_PARM fs_params = {FM_ANY, 0, 0, 0, 0};
        BYTE work[FF_MAX_SS]; // 工作缓冲区
        res = f_mkfs("", &fs_params, work, sizeof(work));
        if (res != FR_OK) {
            // printf("Failed to create filesystem: %d\n", res);
            return;
        }

        // 重新挂载文件系统
        res = f_mount(&fs, "", 1);
        if (res != FR_OK) {
            // printf("Failed to mount filesystem after creation: %d\n", res);
            return;
        }
    }

    // 创建目录“RTCData”
    res = f_mkdir("RTCData");
    if (res != FR_OK && res != FR_EXIST) {
        // printf("Failed to create directory: %d\n", res);
        return;
    }
}
```

这段初始化的逻辑其实相当的简单，只是通过尝试挂载文件系统、处理挂载失败的情况、创建文件系统以及创建特定目录，确保了文件系统的初始化和可用性。这种结构化的错误处理方式使得代码在面对不同情况时能够做出相应的反应，确保系统的稳定性和可靠性。

4.2 FATFS 简单任务

```
// 将 timeStr 放入 RTC.txt 文件中void FATFSTask() {
    UINT bw;
    char timeStr[12];
          //        int i = sizeof(file);
          //        printf("Size of file structure: %d bytes\n", i);
    while (1) {
        // 打开文件
        res = f_open(&file, "RTCData/RTC.txt", FA_WRITE | FA_OPEN_ALWAYS);
        if (res != FR_OK) {
            OSTimeDlyHMSM(0, 0, 1, 0);
            continue;
        }

        // 移动到文件末尾
        res = f_lseek(&file, f_size(&file));
        if (res != FR_OK) {
            f_close(&file);
            OSTimeDlyHMSM(0, 0, 1, 0);
            continue;
        }
        while (1) {
            // 获取时间并格式化
            sprintf(timeStr, "%02u:%02u:%02u\n", time.hour, time.minute, time.second);
            // 写入文件
            res = f_write(&file, timeStr, strlen(timeStr), &bw);
            if (res != FR_OK || bw != strlen(timeStr)) {
                break;
            }
            // 刷新文件缓冲区
            res = f_sync(&file);
            if (res != FR_OK) {
                break;
            }
            // 延迟 1 秒
            OSTimeDlyHMSM(0, 0, 1, 0);
        }
        // 确保文件关闭
         f_close(&file);
        // 延迟 1 秒后重试
        OSTimeDlyHMSM(0, 0, 1, 0);
    }
}
```

首先，定义了一个名为 FATFSTask 的函数，目的是执行文件操作。函数内部声明了一个整型变量 bw，用于存储实际写入文件的字节数。此外，定义了一个字符数组 timeStr，用于存储格式化后的时间字符串。

函数内部包含一个无限循环，表示该任务将持续运行。第一步是尝试打开指定路径的文件。如果文件打开失败，程序会调用延迟函数，暂停一段时间后继续尝试打开文件。这种设计确保了在文件无法访问时不会导致程序崩溃，而是会定期重试。

一旦成功打开文件，程序会将文件指针移动到文件的末尾，以便在文件末尾追加新的数据。如果移动文件指针失败，程序会关闭文件并再次延迟一段时间，然后重试打开文件。这种处理方式确保了文件操作的稳定性。

在成功打开并定位文件后，程序进入另一个无限循环，负责获取当前时间并将其写入文件。时间以“小时:分钟:秒”的格式存储在 timeStr 中。接着，程序尝试将这个字符串写入文件，并检查写入是否成功。如果写入失败，程序会跳出当前循环，准备关闭文件。

在成功写入时间后，程序会调用刷新函数，确保所有缓冲区中的数据都被写入到存储介质中。如果刷新操作失败，程序同样会跳出当前循环，准备关闭文件。

在每次成功写入时间后，程序会延迟一秒钟，以避免频繁写入造成的资源浪费。完成写入后，程序确保文件被正确关闭，以释放资源。然后，程序再次延迟一秒钟，为下一次循环做准备。

4.3 代码运行结果

运行结果如下图：

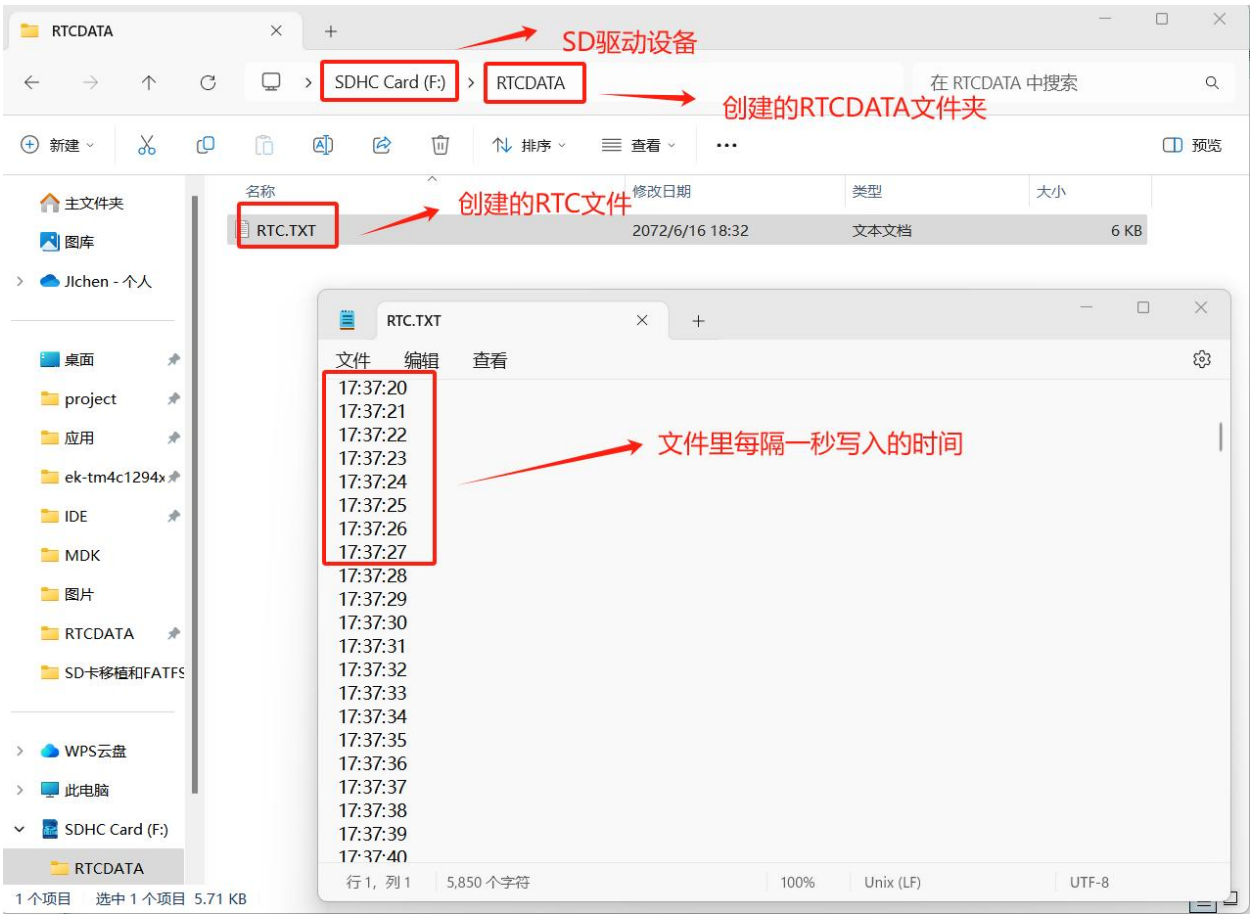

图 5 运行结果图

第五章 常见问题与解决方案

5.1 Fatfs 文件系统的堆栈溢出问题

f_open 函数出现第一次可以调用，第二次调用时出现程序死掉的现象。
解决方案：两处错误。
1. 一开始写的代码忘记放上无限循环，导致代码只允行了一次：

```
// 将timeStr放入RTC.txt文件中
void FATFSTask() {
    // 格式化时间字符串
    sprintf(timeStr, "%02u:%02u:%02u", time.hour, time.minute, time.second);

    // 打开文件"RTCData/RTC.txt"
    FIL file;
    FRESULT res = f_open(&file, "RTCData/RTC.txt", FA_WRITE | FA_OPEN_EXISTING);
    if (res != FR_OK) {
        // printf("Failed to open file: %d\n", res);
        return;
    }

    // 写入时间字符串
    UINT bw;
    res = f_write(&file, timeStr, strlen(timeStr), &bw);
    if (res != FR_OK || bw != strlen(timeStr)) {
        // printf("Failed to write to file: %d\n", res);
        f_close(&file);
        return;
    }

    // 关闭文件
    f_close(&file);

    OSTimeDlyHMSM(0, 0, 1, 0);
}
```

图 6 错误示例 1

2. FATFS FIL 这几个结构体都特别大，需要声明在函数外部：

```
FATFS fs;
FIL file;
FRESULT res;

// 初始化，挂载文件系统
void InitFileSystem() {
```

图 7 真确位置

总结一下当时修改 bug 的思路，以后遇到类似 bug 不会毫无头绪。
第一步：逐个关闭进程，查找死机位置（结果：锁定进程 FATFSTask）
第二步：总结可能出现的原因和方式：
① 内存溢出（解决方法：增加内存；重构代码逻辑）
② 硬件问题（换一个板子试试）
③ 监测程序不到位（增加看门狗程序）
在这个问题中，我首先考虑的就是内存溢出，有可能两个原因：
① 文件操作错误导致资源消耗（结果：增加锁后问题依旧）
② 内部代码问题（逐个关闭函数，观察进程情况）（结果发现 f_open()阻塞）
第三步：精确定位，查找原因
在网上查查可能问题，找到了解决方法。