

STM32Cube高效开发教程（高级篇）

第12章 FatFS文件系统

王维波

中国石油大学（华东）控制科学与工程学院

STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

如果有读者需要本书课件的PPT版本用于备课，可以给作者发邮件免费获取，并可加入专门的教学和技术交流QQ群

邮箱：wangwb@upc.edu.cn



第12章 FatFS文件系统

12.1 FatFS概述

12.2 FatFS的应用程序接口函数

12.3 FatFS的存储介质访问函数

12.4 针对SPI-Flash芯片移植FatFS

12.5 在SPI-Flash芯片上使用文件系统

12.1 FatFS概述

12.1.1 FatFS的作用

12.1.2 文件系统的一些基本概念

12.1.3 FatFS功能特点和参数

12.1.4 FatFS的文件组成

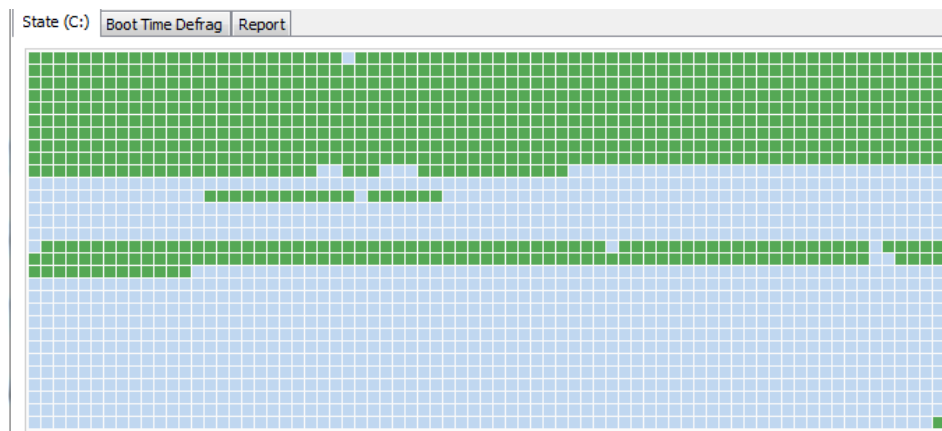
12.1.5 FatFS的基本数据类型定义

12.1.1 FatFS的作用

文件系统是管理存储介质上的文件的软件系统。通过目录、文件管理数据

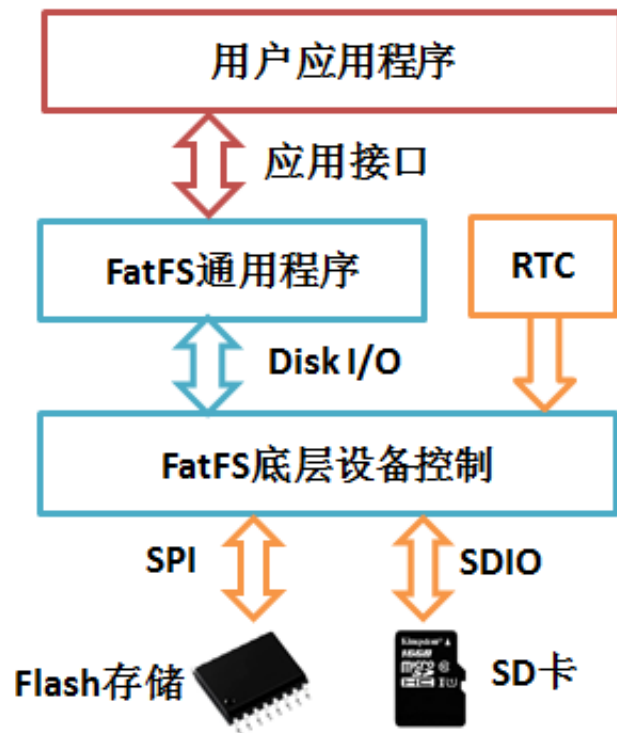
在没有文件系统的存储器中读写数据时就需要进行底层的操作。例如SPI接口的Flash存储芯片W25Q128。它有16M字节存储空间，分为256个块、4096个扇区和65536个页。

名称	类型	大小
.settings	文件夹	
Debug	文件夹	
Drivers	文件夹	
Inc	文件夹	
Middlewares	文件夹	
Src	文件夹	
Startup	文件夹	
.cproject	CPROJECT 文件	27 KB
.mxproject	MXPROJECT ...	9 KB
.project	PROJECT 文件	2 KB
Demo12_1FlashFAT Deb...	LAUNCH 文件	7 KB
Demo12_1FlashFAT.ioc	STM32CubeMX	11 KB
STM32F407ZGTX_FLASH....	LD 文件	6 KB
STM32F407ZGTX_RAM.ld	LD 文件	6 KB



FatFS (FAT File System) 是一个用于嵌入式系统的文件管理系统，它可以在SD卡、U盘、Flash芯片等存储介质上创建FAT或exFAT文件系统，它提供应用层接口函数使用户能直接通过文件进行数据读写操作和管理。

FatFS完全用ANSI C语言编写，文件操作的软件模块与底层硬件访问层完全分离，能方便地移植到各种处理器平台和存储介质。



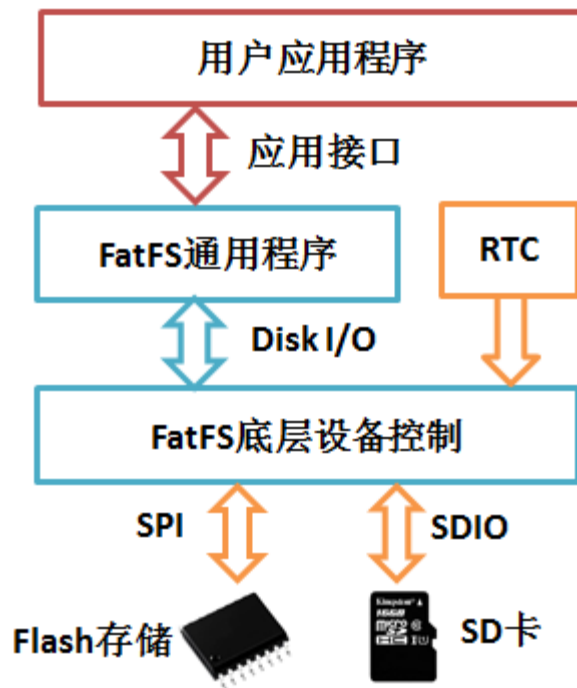
使用FatFS的嵌入式系统软硬件结构

(1) 用户应用程序。

通过FatFS的通用接口API函数进行文件系统的操作，例如用函数f_open()打开一个文件，用函数f_write()向文件写入数据，用函数f_close()关闭一个文件。

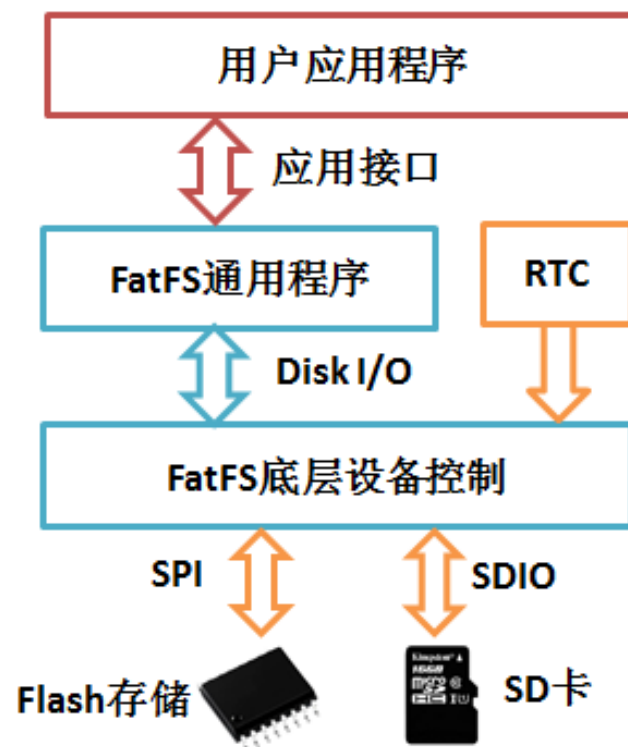
(2) FatFS通用程序。

与硬件无关的用于文件系统管理的一些通用操作API函数，包括文件系统的创建和挂载、目录操作和文件操作等函数。



(3) FatFS底层设备控制

FatFS与底层存储设备通讯的一些功能函数，包括读写存储介质的函数 `disk_read()` 和 `disk_write()` 等。FatFS底层设备控制部分是与硬件密切相关的，FatFS的移植主要就这部分的移植



(4) 存储介质和RTC

例如Flash存储芯片、SD卡、U盘、扩展的SRAM等。在一个嵌入式设备上可以使用FatFS管理多个存储设备。RTC用于获取当前时间，作为文件的时间戳信息

12.1.2 文件系统的一些基本概念

1. 文件系统

FatFS可以在存储介质上创建和管理FAT或exFAT文件系统，这两种都是Microsoft公司定义的标准文件系统。

FAT (File Allocation Table) 文件系统起源于1980在MS-DOS中使用的文件系统，现在有3种FAT文件系统：FAT12、FAT16和FAT32。其中，

- FAT16的单个分区容量不能超过2GB；
- FAT32的单个分区容量不能超过32GB，单个文件大小不能超过4GB

exFAT (Extended File Allocation Table)是继FAT16/FAT32之后Microsoft开发的一种文件系统，它更适用于基于闪存的存储器，如SD卡、U盘等，而不适用于机械硬盘。

exFAT广泛应用于嵌入式系统、消费电子产品和固态存储设备中。

- exFAT文件系统允许单个文件大小超过4GB
- 单个分区大小和单个文件大小几乎没有限制

注意：FatFS不支持NTFS文件系统

2. FAT卷

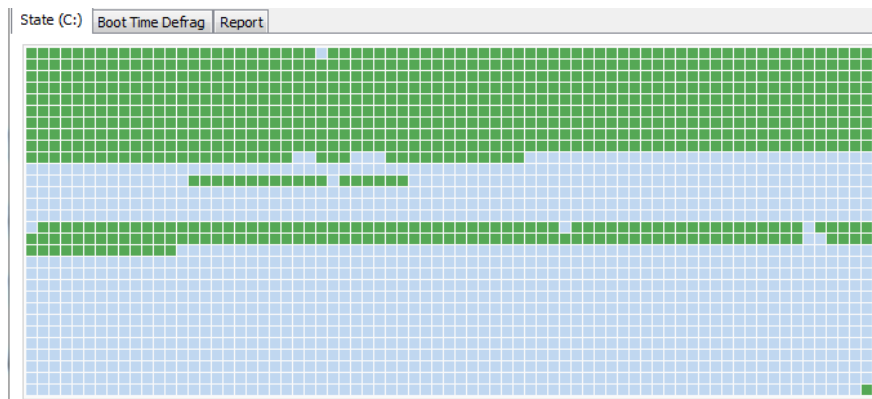
一个FAT文件系统称为一个**逻辑卷**（logical volume）或**逻辑驱动器**（logical drive），如电脑上的C盘、D盘。一个FAT卷包括如下的3个或4个区域（Area），每个区域占用1个或多个扇区，并按如下的顺序排列：

- **保留区域**，用于存储卷的配置数据。
- **FAT区域**，用于存储数据区的分配表。
- **根目录区域**，在FAT32卷上没有这个区域。
- **数据区域**，存储文件和目录的内容。

3. 扇区

扇区（Sector）是存储介质上读写数据的最小单元。一般的扇区大小是512字节，FatFS支持512、1024、2048和4096字节等几种大小的扇区。

存储设备上的每个扇区有一个扇区编号，从设备的起始位置开始编号的称为物理扇区号（physical sector number），也就是扇区的绝对编号。另外也可以从卷的起始位置开始相对编号，称为扇区号（sector number）



4. 簇

一个卷的数据区分为多个簇（cluster），一个簇包含1个或多个扇区，数据区就是以簇为单位进行管理的。一个卷的FAT类型就是由其包含的簇的个数决定的，由簇的个数就可以判断卷的FAT类型。

- FAT12卷，簇的个数 ≤ 4085
- FAT16卷， $4086 \leq \text{簇的个数} \leq 65525$
- FAT32卷，簇的个数 ≥ 65526

12.1.3 FatFS功能特点和参数

FatFS有如下的特性和限制：

- 支持FAT和exFAT文件系统
- 支持长文件名和Unicode
- 对于RTOS是线程安全的
- 最多可以管理10个卷，可以是多个存储介质或多个分区
- 支持不同大小的扇区，扇区大小可以是512、1024、2048、4096字节
- 同时打开的文件个数：无限制，只受限于内存大小
- 最小卷大小：128个扇区
- 最大卷大小：FAT中是4G个扇区，exFAT中几乎是无限限制的
- 最大单个文件大小：FAT卷中是4GB，exFAT中几乎是无限限制的
- 簇大小限制：FAT卷中一个簇最大128个扇区，exFAT卷中是16MB

12.1.4 FatFS的文件组成

以SPI-Flash存储芯片W25Q128为例，图中的箭头表示调用关系

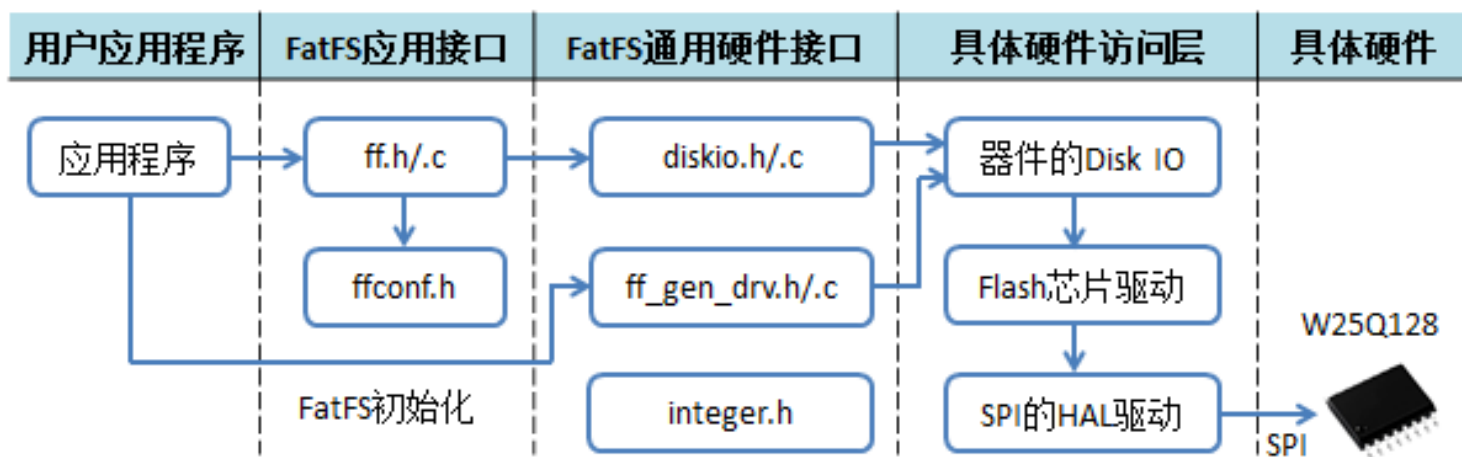
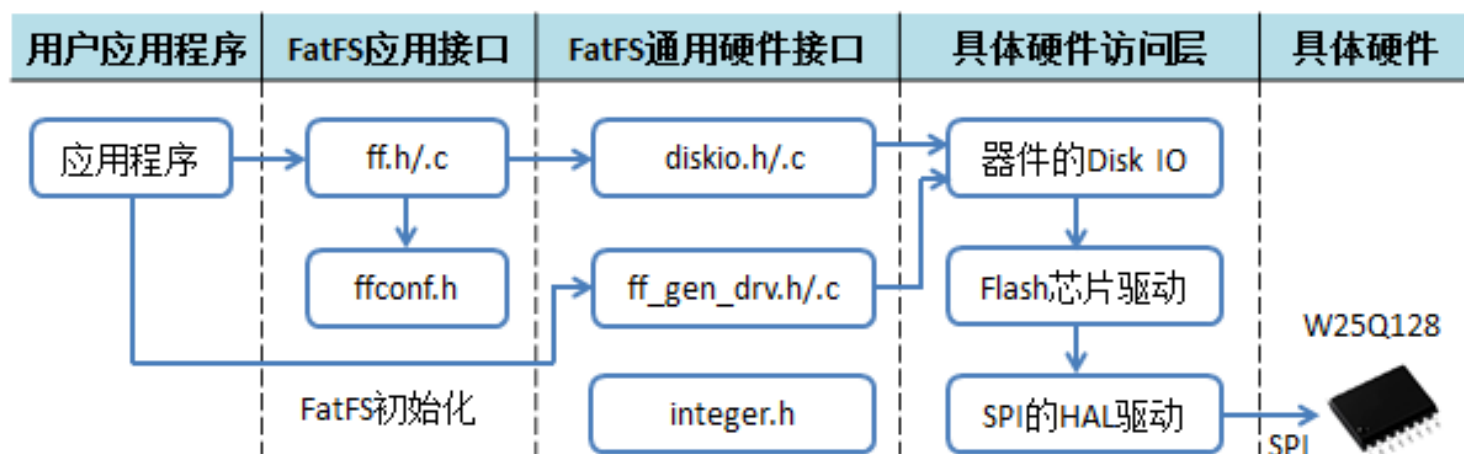


图12-2 使用FatFS时的文件分层结构

1. 用户应用程序

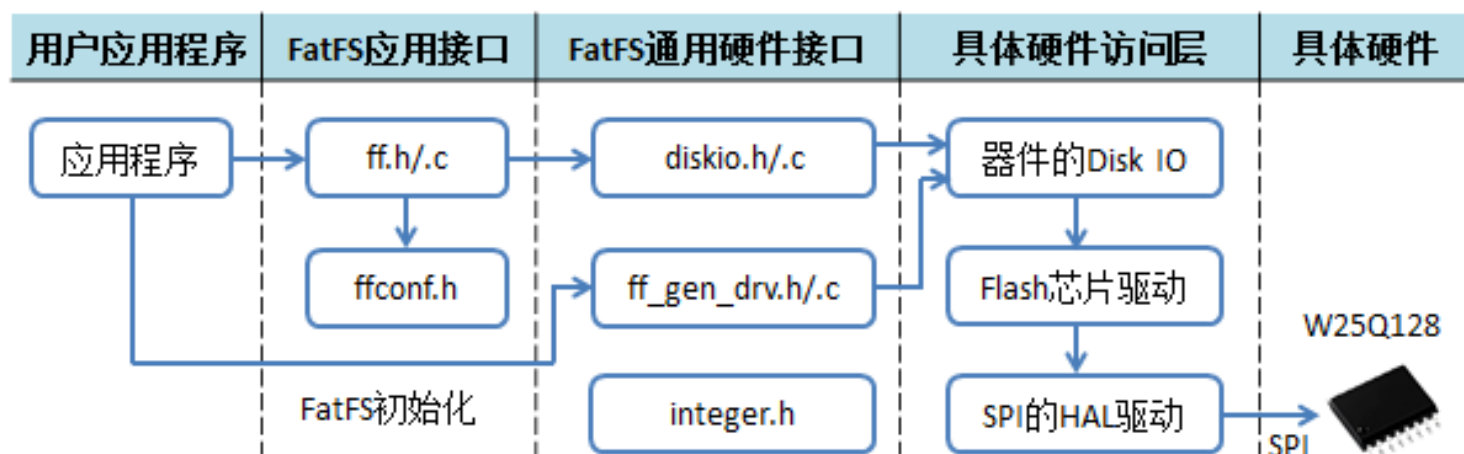
包括CubeMX自动生成的FatFS初始化程序和用户编写的文件系统访问程序。用户程序就是使用文件`ff.h`中提供的API函数进行文件系统操作，例如用`f_open()`打开文件，`f_write()`写入数据等



2. FatFS应用接口

这是面向用户应用程序的编程接口，提供文件操作的API函数，这些API函数与具体的存储介质和处理器类型无关。

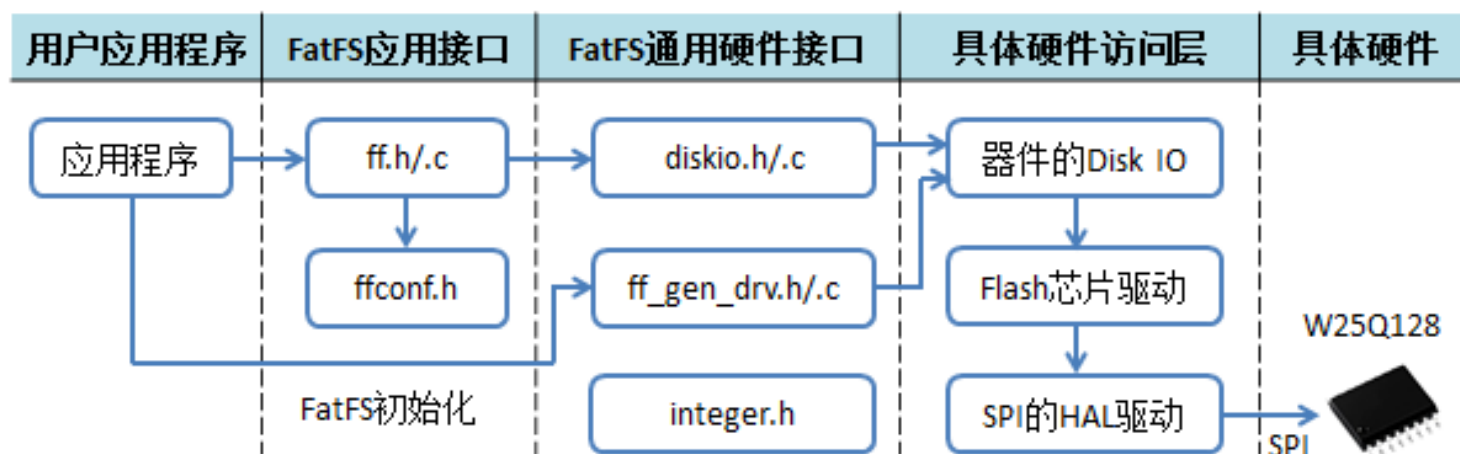
文件ff.h和ff.c中定义和实现了这些API函数。文件ffconf.h是FatFS的配置文件，用于定义FatFS的一些参数，以便进行功能裁剪。



3. FatFS通用硬件接口

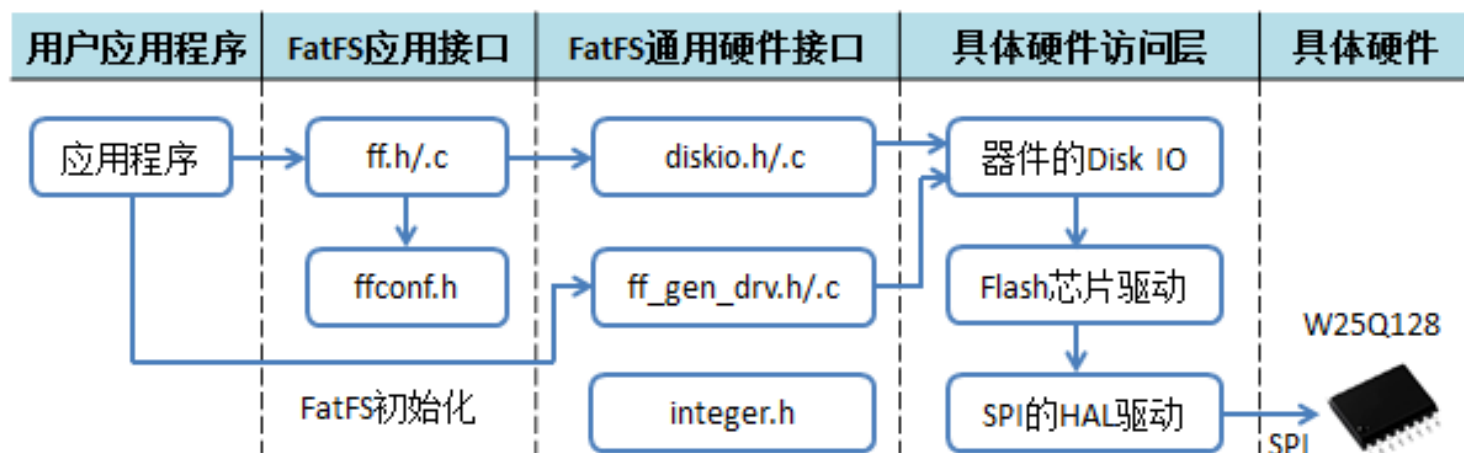
实现存储介质访问（Disk IO）的通用接口。文件diskio.h/.c中定义和实现了Disk IO的几个基本函数，包括disk_initialize()、disk_read()、disk_ioctl()等。

文件ff_gen_drv.h中定义了驱动器和驱动器列表管理的一些类型和函数。主程序里进行FatFS初始化时会将一个驱动器链接到FatFS管理的驱动器列表里。



4. 具体硬件访问层

假设一个存储介质只建立一个逻辑分区，那么一个存储介质就是一个驱动器，例如一个SD卡，或一个U盘。一个驱动器需要实现自己的Disk IO函数，驱动器会通过函数指针将diskio.h中定义的通用Disk IO函数指向器件的Disk IO函数。器件的Disk IO函数是在一个特定的文件里实现的，而这些Disk IO函数又会调用器件的驱动程序或硬件接口的HAL驱动程序。



5. 具体硬件

在嵌入式设备中，一个存储介质一般只有一个分区

SPI-Flash存储芯片W25Q128就是用户定义器件

FATFS Mode and Configuration	
Mode	
<input type="checkbox"/>	External SRAM
<input type="checkbox"/>	SD Card
<input type="checkbox"/>	USB Disk
<input type="checkbox"/>	User-defined

图12-3 CubeMX中FatFS支持的存储介质类型

12.1.5 FatFS的基本数据类型定义

文件interger.h对基本数据类型重新定义了符号，便于在不同的处理器平台上移植

```
/* These types MUST be 16-bit or 32-bit */
```

```
typedef int INT;
```

```
typedef unsigned int UINT;
```

```
/* This type MUST be 8-bit */
```

```
typedef unsigned char BYTE;
```

```
/* These types MUST be 16-bit */
```

```
typedef short SHORT;
```

```
typedef unsigned short WORD;
```

```
typedef unsigned short WCHAR;
```

```
/* These types MUST be 32-bit */
```

```
typedef long LONG;
```

```
typedef unsigned long DWORD;
```

```
/* This type MUST be 64-bit (Remove this for ANSI C (C89) compatibility) */
```

```
typedef unsigned long long QWORD;
```

12.2 FatFS的应用程序接口函数

12.2.1 卷管理和系统配置相关函数

12.2.2 文件和目录管理相关函数

12.2.3 目录访问相关函数

12.2.4 文件访问相关函数

12.2.1 卷管理和系统配置相关函数

注意，官网上某些函数的原型定义与CubeMX生成代码中的实际定义不相同。

函数名	函数功能	函数原型
f_mount	注册或注销一个卷, 使用一个卷之前必须调用此函数挂载文件系统	f_mount(FATFS* fs, const TCHAR* path, BYTE opt)
f_mkfs	在一个逻辑驱动器上创建FAT卷, 也就是进行格式化	f_mkfs(const TCHAR* path, BYTE opt, DWORD au, void* work, UINT len)
f_fdisk	在一个物理驱动器上创建分区	f_fdisk(BYTE pdrv, const DWORD* szt, void* work)
f_getfree	返回一个卷上剩余簇的个数	f_getfree(const TCHAR* path, DWORD* nclst, FATFS** fatfs)
f_getlabel	获取一个卷的标签	f_getlabel(const TCHAR* path, TCHAR* label, DWORD* vsn)
f_setlabel	设置一个卷的标签	f_setlabel(const TCHAR* label)

函数f_mount()

FatFS要求每个逻辑驱动器（FAT卷）有一个文件系统对象作为工作区域，在对这个逻辑驱动器进行文件操作之前，需要用函数f_mount()将逻辑驱动器和文件系统对象注册。

```
FRESULT f_mount (  
    FATFS* fs,           /* [IN]文件系统对象指针，NULL表示卸载 */  
    const TCHAR* path, /* [IN]需要挂载或卸载的逻辑驱动器号，如"0:" */  
    BYTE opt             /* [IN]模式选项， 0: 延迟挂载, 1:立刻挂载 */  
)
```

如果f_mount()返回值为FR_NO_FILESYSTEM，表示存储介质还没有文件系统，需要调用f_mkfs()函数格式化

```
FATFS fs;                //文件系统对象
FRESULT res= f_mount(&fs, "0:", 1);    //立刻挂载文件系统
if (res==FR_NO_FILESYSTEM)    //不存在文件系统，未格式化
{
    BYTE workBuffer[4096];        //工作缓冲区
    res=f_mkfs("0:", FM_FAT, 4096, workBuffer, 4096); //簇大小=4096字节
    if (res ==FR_OK)                //格式化成功
    {
        res=f_mount(NULL, "0:", 1);    //先卸载
        res=f_mount(&fs, "0:", 1); //再次挂载文件系统
    }
}
```


12.2.2 文件和目录管理相关函数

函数名	函数功能	函数原型
f_stat	检查一个文件或目录是否存在	f_stat(const TCHAR* path, FILINFO* fno)
f_unlink	删除一个文件或目录	f_unlink(const TCHAR* path)
f_rename	重命名或移动一个文件或目录	f_rename(const TCHAR* path_old, const TCHAR* path_new)
f_chmod	改变一个文件或目录的属性	f_chmod(const TCHAR* path, BYTE attr, BYTE mask);
f_utime	改变一个文件或目录的时间戳	f_utime(const TCHAR* path, const FILINFO* fno)
f_mkdir	创建一个新的目录	f_mkdir(const TCHAR* path);
f_chdir	改变当前工作目录	f_chdir(const TCHAR* path);
f_chdrive	改变当前驱动器	f_chdrive(const TCHAR* path)
f_getcwd	获取当前驱动器的当前工作目录	f_getcwd(TCHAR* buff, UINT len)

12.2.3 目录访问相关函数

函数名	函数功能	函数原型
f_opendir	打开一个已经存在的目录	f_opendir(DIR* dp, const TCHAR* path)
f_closedir	关闭一个打开的目录	f_closedir(DIR *dp)
f_readdir	读取目录下的一个项	f_readdir(DIR* dp, FILINFO* fno)
f_findfirst	在目录下查找第一个匹配项	f_findfirst(DIR* dp, FILINFO* fno, const TCHAR* path, const TCHAR* pattern)
f_findnext	查找下一个匹配项	f_findnext(DIR* dp, FILINFO* fno)

使用目录访问函数可以打开一个目录，然后查找这个目录下匹配的文件或目录。可以列出一个目录下的特定类型文件。

12.2.4 文件访问相关函数

函数名	函数功能	函数原型
f_open	打开一个文件	f_open(FIL* fp, const TCHAR* path, BYTE mode)
f_close	关闭一个打开的文件	f_close(FIL* fp)
f_read	从文件读取数据	f_read(FIL* fp, void* buff, UINT btr, UINT* br)
f_write	将数据写入文件	f_write(FIL* fp, const void* buff, UINT btw, UINT* bw)
f_lseek	移动读写操作的指针	f_lseek(FIL* fp, FSIZE_t ofs)
f_truncate	截断文件	f_truncate(FIL* fp)
f_sync	将缓存的数据写入文件	f_sync(FIL* fp)
f_forward	读取数据直接传给数据流设备	f_forward(FIL* fp, UINT(*func)(const BYTE*,UINT), UINT btf, UINT* bf)

文件访问相关函数（续表）

函数名	函数功能	函数原型
f_expand	为文件分配连续的存储空间	f_expand(FIL* fp, FSIZE_t szf, BYTE opt)
f_gets	从文件读取一个字符串	TCHAR* f_gets(TCHAR* buff, int len, FIL* fp)
f_putc	向文件写入一个字符	int f_putc(TCHAR c, FIL* fp);
f_puts	向文件写入一个字符串	int f_puts(const TCHAR* str, FIL* cp)
f_printf	用格式写入字符串	int f_printf(FIL* fp, const TCHAR* str, ...);
f_tell	获取当前的读写指针	#define f_tell(fp) ((fp)->fptr)
f_eof	检测是否到文件尾端	#define f_eof(fp) ((int)((fp)->fptr == (fp)->obj.objsize))
f_size	获取文件大小，单位：字节	#define f_size(fp) ((fp)->obj.objsize)
f_error	检测是否有错误，返回0表示无错误	#define f_error(fp) ((fp)->err)

1. 函数f_open()

函数f_open()用于打开或新建一个文件

```
FRESULT f_open(  
    FIL* fp,                /* [OUT] 文件对象指针 */  
    const TCHAR* path,      /* [IN] 文件名 */  
    BYTE mode               /* [IN] 访问模式 */  
)
```

参数mode是文件访问模式，是一些宏定义的位运算组合，
这些模式的宏定义如下：

#define FA_READ	0x01	//读取模式
#define FA_WRITE	0x02	//写入模式, FA_READ FA_WRITE表示可读可写
#define FA_OPEN_EXISTING	0x00	//打开的文件必须已存在，否则函数失败
#define FA_CREATE_NEW	0x04	//新建文件，如果文件已经存在，函数失败并返回FR_EXIST
#define FA_CREATE_ALWAYS	0x08	//总是新建文件，如果文件已经存在会覆盖现有文件
#define FA_OPEN_ALWAYS	0x10	//打开一个已存在的文件，如果不存在就创建新文件
#define FA_OPEN_APPEND	0x30	//与FA_OPEN_ALWAYS相同，只是读写指针定位在文件尾端

2. 函数f_write()

函数f_write()用于向一个以可写方式打开的文件写入数据

```
FRESULT f_write (  
    FIL* fp,                /* [IN] 文件对象指针 */  
    const void* buff,       /* [IN] 待写入的数据缓存区指针 */  
    UINT btw,               /* [IN] 待写入数据的字节数 */  
    UINT* bw                /* [OUT] 实际写入文件的数据字节数 */  
)
```

参数fp是用f_open()打开文件时返回的文件对象指针。文件对象结构体FIL有一个DWORD类型的成员变量fptr，称为文件读写位置指针，它表示文件内当前读写位置。文件打开后，这个读写位置指针指向文件顶端。f_write()在当前的读写位置指针处向文件写入数据，写入数据后读写位置指针会自动向后移动。

代码段：用f_open()新建一个文件，然后向文件写入一些数据，写完数据后用f_close()关闭文件。

```
void TestWriteBinFile( TCHAR *filename, uint32_t pointCount, uint32_t sampFreq)
{
    FIL file;
    FRESULT res=f_open(&file,filename, FA_CREATE_ALWAYS | FA_WRITE);
    if(res == FR_OK)
    {
        UINT bw=0; //实际写入字节数
        f_write(&file, &pointCount, sizeof(uint32_t), &bw); //数据点个数
        f_write(&file, &sampFreq, sizeof(uint32_t), &bw); //采样频率
        uint32_t value=1000;
        for(uint16_t i=0; i<pointCount; i++,value++)
            f_write(&file, &value, sizeof(uint32_t), &bw);
    }
    f_close(&file);
}
```

3. 函数f_read()

函数f_read ()用于从一个可读文件中读取数据

```
FRESULT f_read(  
    FIL* fp,                /* [IN] 文件对象指针 */  
    void* buff,             /* [OUT] 保存读出数据的缓存区 */  
    UINT btr,               /* [IN] 要读取数据的字节数 */  
    UINT* br                /* [OUT] 实际读取数据的字节数 */  
)
```

f_read()会从文件的当前读写位置处读取btr个字节的数据，读出的数据保存到缓存区buff里，实际读取的字节数返回到变量*br里。读出数据后，文件的读写指针会自动向后移动。

与前面的函数TestWriteBinFile()对应的，从一个文件读取数据的代码如下：

```
void TestReadBinFile( TCHAR *filename)
{
    FIL file;
    FRESULT res=f_open(&file,filename, FA_READ);
    if(res == FR_OK)
    {
        UINT bw=0; //实际读取字节数
        uint32_t pointCount, sampFreq;
        f_read(&file, &pointCount, sizeof(uint32_t), &bw); //数据点个数
        f_read(&file, &sampFreq, sizeof(uint32_t), &bw); //采样频率
        uint32_t value;
        for(uint16_t i=0; i< pointCount; i++)
            f_read(&file, &value, sizeof(uint32_t), &bw);
    }
    f_close(&file);
}
```

4. 读写字符串的函数f_puts()和f_gets()

f_write()和f_read()是通用的数据读写函数，但不适合于读写字符串。f_puts()用于写入一个字符串，函数原型为：

```
int f_puts (const TCHAR* str, FIL* fp)
```

str是以'\0'作为结束符的字符串指针，fp是文件对象指针。如果写入成功，函数返回值是实际写入的字节数，否则返回-1

```
TCHAR str[]="Line1: Hello, FatFS\n";    //字符串必须有换行符'\n'  
f_puts(str, &file);    //不会写入结束符'\0'
```

注意，字符串str必须在末尾加上换行符'\n'，使用f_puts()将字符串写入文件时，不会将字符串的结束符'\0'写入文件。

函数f_gets()用于读取一个字符串，它通过换行符'\n'判断一个字符串的结束，在读出的字符串末尾会自动添加结束符'\0'

```
TCHAR* f_gets (TCHAR* buff, int len, FILE* fp)
```

参数buff是保存读出字符串的缓存区，len是buff的长度，fp是文件对象指针。这个函数的使用示例代码如下：

```
TCHAR str[100];  
f_gets(str, 100, &file);
```

注意，这里的第二个参数值100是指缓存区str的长度，而不是实际读出的字符串的长度。缓存区str的长度要足够大，能容纳一次读取的一行字符串。

5. 文件内读写指针的移动

- `f_tell()`返回文件读写指针的当前值
- `f_lseek()`直接将文件读写指针移动到文件内的某个绝对位置
- `f_eof()`可以判断文件读写指针是否到文件尾端了

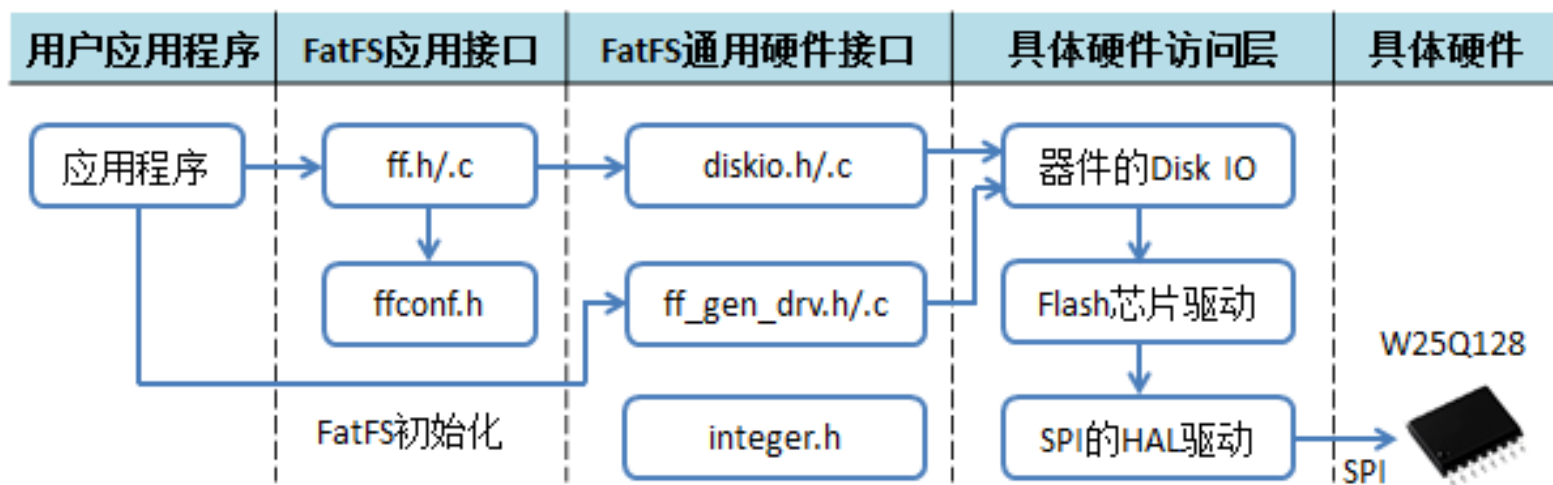
例如读取一个全是字符串的文本文件内容，并在LCD上显示的示意代码如下：

```
FIL file;
FRESULT res=f_open(&file,"readme.txt", FA_READ);
if(res == FR_OK)
{
    TCHAR str[100];
    while (!f_eof(&file))
    {
        f_gets(str,100, &file);    //读取1个字符串
        LCD_ShowStr(10,LCD_CurY+20, (uint8_t *)str);
    }
}
```

12.3 FatFS的存储介质访问函数

硬件层的访问接口在文件diskio.h中定义。FatFS的移植主要就是针对存储介质的硬件层访问程序的移植。

FatFS已经尽量简化了硬件访问层函数的定义，并且使用了统一的函数接口，移植时只需重新实现这些函数即可。



硬件层访问相关函数

函数名	函数功能
disk_status	获取驱动器当前状态，例如是否已完成初始化
disk_initialize	设备初始化，就是硬件接口的初始化
disk_read	读取一个或多个扇区的数据
disk_write	数据写入一个或多个扇区，当_USE_WRITE==1时才需实现这个函数
disk_ioctl	设备IO控制，例如获取扇区个数、扇区大小等参数。当_USE_IOCTL == 1时才需实现这个函数
get_fattime	获取当前时间作为文件的修改时间，可以通过RTC获取当前时间，也可以不实现这个函数

前3个函数是必须实现的。存储设备数据读写的最小单位是1个扇区，`disk_read()`读取一个或多个扇区的数据到缓存区，`disk_write()`将数据写入一个或多个扇区。

12.4 针对SPI-Flash芯片移植FatFS

12.4.1 SPI-Flash芯片硬件电路

12.4.2 CubeMX项目基础设置

12.4.3 在CubeMX中设置FatFS

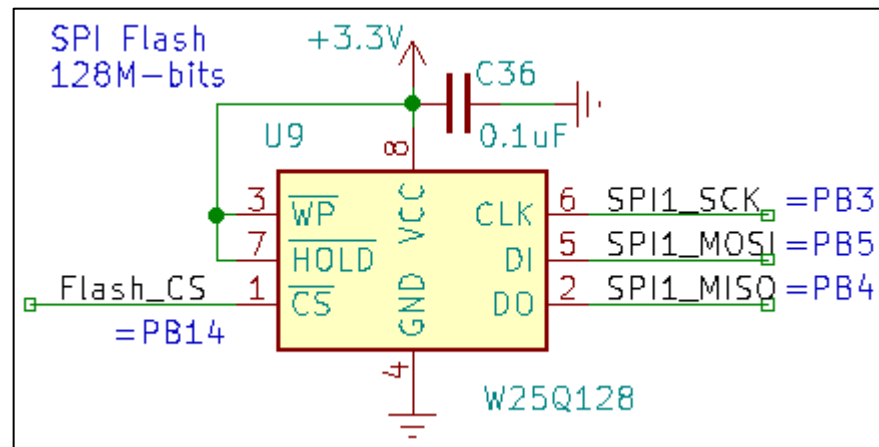
12.4.4 项目中FatFS的文件组成

12.4.5 FatFS初始化过程

12.4.6 针对SPI-Flash芯片的Disk IO函数实现

12.4.1 SPI-Flash芯片硬件电路

Flash存储芯片W25Q128，
与MCU的SPI1接口连接。总容量是16M字节，参数如下：



- 总共256个块（Block），每个块64K字节
- 每个块又分为16个扇区（Sector），共4096个扇区，每个扇区4K字节
- 每个扇区又分为16个页（Page），共65536个页，每个页256字节

12.4.2 CubeMX项目基础设置

1. RTC设置

启用RTC，用于为FatFS提供时间数据

2. SPI1设置

CubeMX会自动计算波特率，波特率为6.25 Mbps时通讯比较稳定。

数据传输是MSB先行，
CPOL和CPHA的组合是SPI
时序模式3，与W25Q128的
SPI接口参数一致。

Parameter Settings		User Constants
Basic Parameters		
Frame Format		Motorola
Data Size		8 Bits
First Bit		MSB First
Clock Parameters		
Prescaler (for Baud Rate)		8
* Baud Rate		6.25 MBits/s
Clock Polarity (CPOL)		High
Clock Phase (CPHA)		2 Edge
Advanced Parameters		
CRC Calculation		Disabled
NSS Signal Type		Software

12.4.3 在CubeMX中设置FatFS

1. FatFS模式设置

模式设置就是选择FatFS应用的存储介质，有4个选项

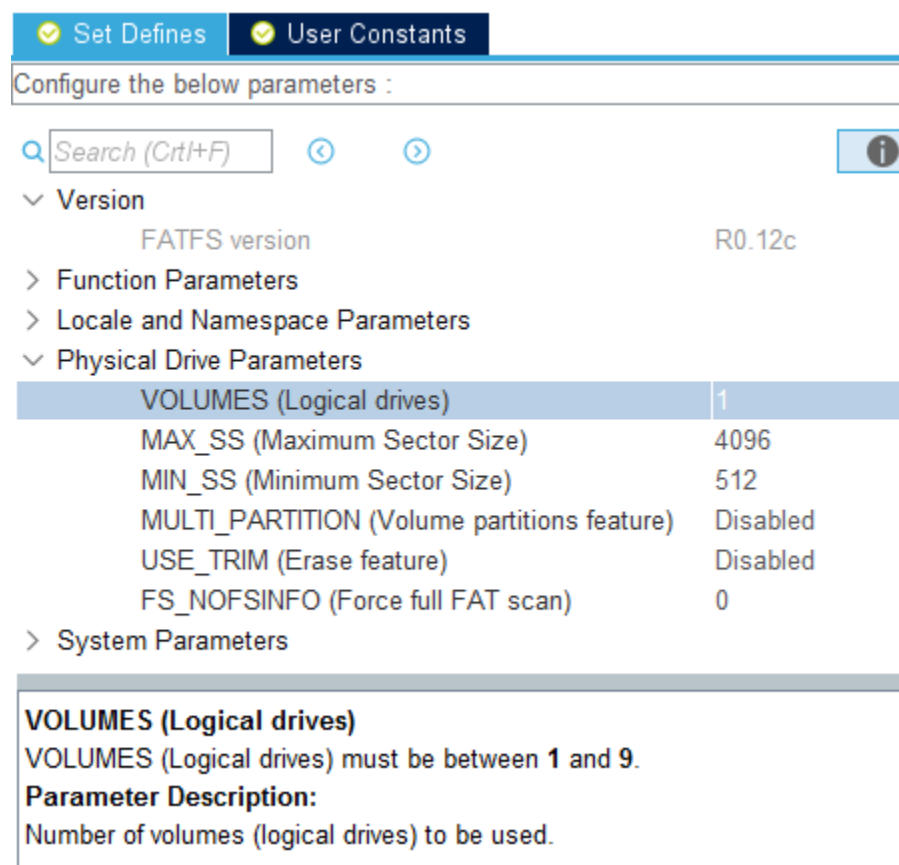
User-defined（用户定义器件），例如开发板上连接在SPI1接口上的Flash存储芯片W25Q128

FATFS Mode and Configuration

Mode
<input type="checkbox"/> External SRAM
<input type="checkbox"/> SD Card
<input type="checkbox"/> USB Disk
<input checked="" type="checkbox"/> User-defined

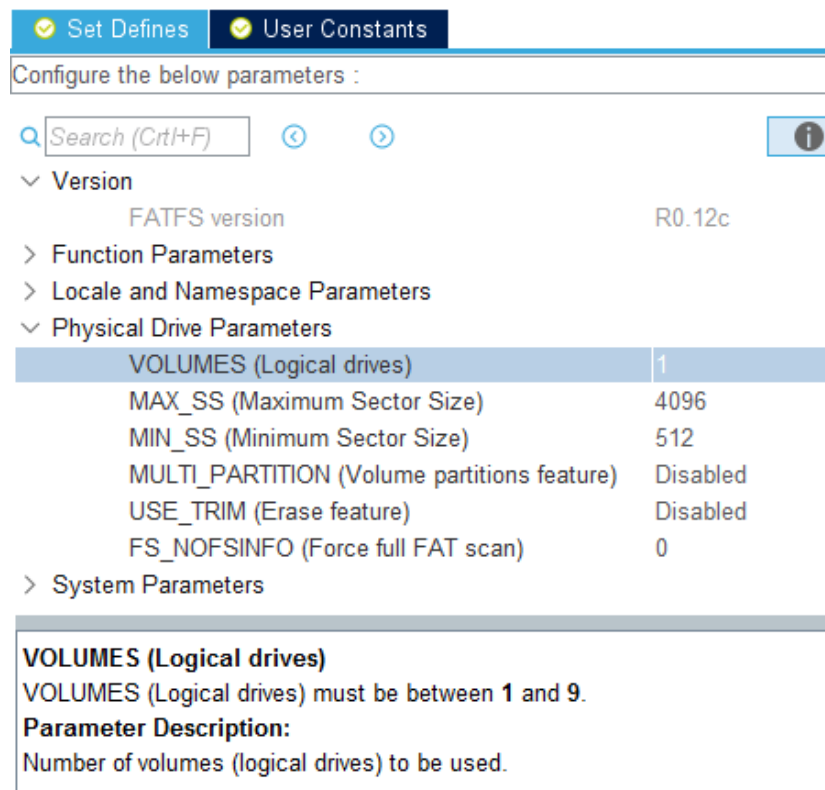
2. FatFS参数设置概述

- (1) Version组, FatFS的版本
- (2) Function Parameters组, 用于配置是否包含某些函数
- (3) Locale and Namespace Parameters组, 本地化和名称空间参数, 例如设置代码页, 是否使用长文件名等
- (4) Physical Driver Parameters组, 物理驱动器参数, 包括卷的个数, 扇区大小等
- (5) System Parameters组, 系统参数, 一些系统级的参数定义



本示例中对FatFS各参数的设置基本保留其默认值，只修改了如下两个参数：

- **CODE_PAGE**，语言代码页，
设置为Simplified
Chinese(DBCS)
- **MAX_SS**，扇区大小最大值，
设置为4096，因为W25Q128
的扇区大小就是4096字节



The screenshot shows the FatFS configuration tool interface. At the top, there are two tabs: 'Set Defines' (checked) and 'User Constants' (checked). Below the tabs, a text box says 'Configure the below parameters :'. There is a search bar with the placeholder 'Search (Ctrl+F)' and two navigation arrows. The main content area is divided into sections: 'Version' (FATFS version: R0.12c), 'Function Parameters', 'Locale and Namespace Parameters', and 'Physical Drive Parameters'. Under 'Physical Drive Parameters', there is a table with the following data:

VOLUMES (Logical drives)	
MAX_SS (Maximum Sector Size)	4096
MIN_SS (Minimum Sector Size)	512
MULTI_PARTITION (Volume partitions feature)	Disabled
USE_TRIM (Erase feature)	Disabled
FS_NOFSINFO (Force full FAT scan)	0

Below the table, there is a section for 'System Parameters' which contains the following text:

VOLUMES (Logical drives)
VOLUMES (Logical drives) must be between 1 and 9.
Parameter Description:
Number of volumes (logical drives) to be used.

3. Function Parameters组参数设置

Function Parameters	
FS_READONLY (Read-only mode)	Disabled
FS_MINIMIZE (Minimization level)	Disabled
USE_STRFUNC (String functions)	Enabled with LF -> CRLF conversion
USE_FIND (Find functions)	Disabled
USE_MKFS (Make filesystem function)	Enabled
USE_FASTSEEK (Fast seek function)	Enabled
USE_EXPAND (Use f_expand function)	Disabled
USE_CHMOD (Change attributes function)	Disabled
USE_LABEL (Volume label functions)	Disabled
USE_FORWARD (Forward function)	Disabled

第一列是参数名称，对应于源文件中的宏，宏的名称就是参数名称前加“_”，例如参数FS_READONLY对应的宏是“_FS_READONLY”

第二列是参数值，这些参数一般是逻辑值，Disabled表示设置为0，Enabled表示设置为1

4. Locale and Namespace Parameters组参数设置

Locale and Namespace Parameters	
CODE_PAGE (Code page on target)	Simplified Chinese (DBCS)
USE_LFN (Use Long Filename)	Disabled
MAX_LFN (Max Long Filename)	255
LFN_UNICODE (Enable Unicode)	ANSI/OEM
STRF_ENCODE (Character encoding)	UTF-8
FS_RPATH (Relative Path)	Disabled

参数	默认值	可设置内容和功能描述
CODE_PAGE	Latin 1	设置目标系统上使用的OEM编码页，编码页如果选择不正常，可能导致打开文件失败。如果要支持中文应该选择Simplified Chinese(DBCS)，对应_CODE_PAGE参数值是936
USE_LFN	Disabled	是否使用长文件名（LFN）
MAX_LFN	255	设定值范围12至255，是LFN的最大长度
LFN_UNICODE	ANSI/OEM	是否将FatFS API中的字符编码切换为Unicode。需要USE_LFN设置为Enabled时，LFN_UNICODE才可以设置为1（Unicode）。当USE_LFN设置为Disabled时，LFN_UNICODE只能设置为ANSI/OEM
STRF_ENCODE	UTF-8	当启用Unicode后，FatFS API 中的字符编码都需要转换为Unicode。这个参数选择字符串操作相关函数读写文件时用的编码
FS_RPATH	Disabled	是否使用相对路径，以及使用相对路径时的特性

5. Physical Driver Parameters组参数设置

Physical Drive Parameters	
VOLUMES (Logical drives)	1
MAX_SS (Maximum Sector Size)	4096
* MIN_SS (Minimum Sector Size)	512
MULTI_PARTITION (Volume partitions feature)	Disabled
USE_TRIM (Erase feature)	Disabled
FS_NOFSINFO (Force full FAT scan)	0

参数	默认值	可设置内容和功能描述
VOLUMES	1	使用的逻辑驱动器的个数，设置范围1到9
MAX_SS	512	最大扇区大小（字节数），只能设置为512、1024、2048或4096，本示例使用的Flash存储芯片W25Q128的扇区大小为4096字节，所以设置为4096。当MAX_SS大于512时，在disk_ioctl()函数中需要实现GET_SECTOR_SIZE指令
MIN_SS	512	最小扇区大小（字节数），只能设置为512、1024、2048或4096
MULTI_PARTITION	Disabled	设置为Disabled时，每个卷与相同编号的物理驱动器绑定，只会挂载第一个分区。设置为Enabled时，每个卷与分区表VolToPart[]关联。
USE_TRIM	Disabled	是否使用ATA_TRIM特性。要想使用Trim特性，在disk_ioctl()函数中应该实现CTRL_TRIM指令
FS_NOFSINFO	0	参数取值0、1、2或3，它设置了函数f_getfree()的运行特性

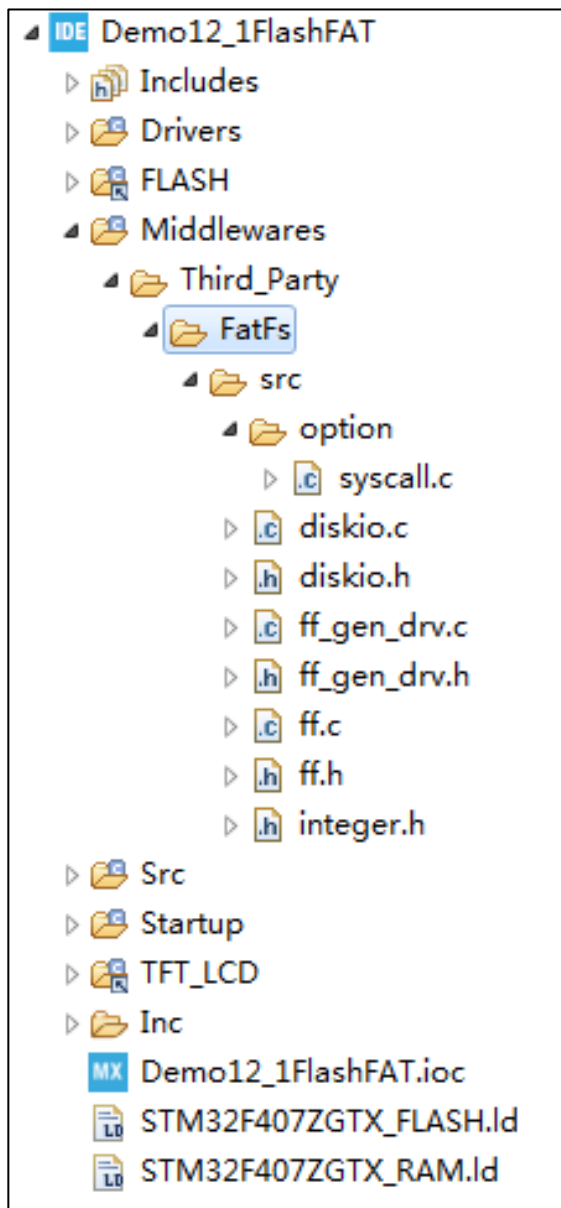
6. System Parameters组参数设置

System Parameters	
FS_TINY (Tiny mode)	Disabled
FS_EXFAT (Support of exFAT file system)	Disabled
FS_NORTC (Timestamp feature)	Dynamic timestamp
FS_REENTRANT (Re-Entrancy)	Disabled
FS_TIMEOUT (Timeout ticks)	1000
FS_LOCK (Number of files opened simultaneously)	2

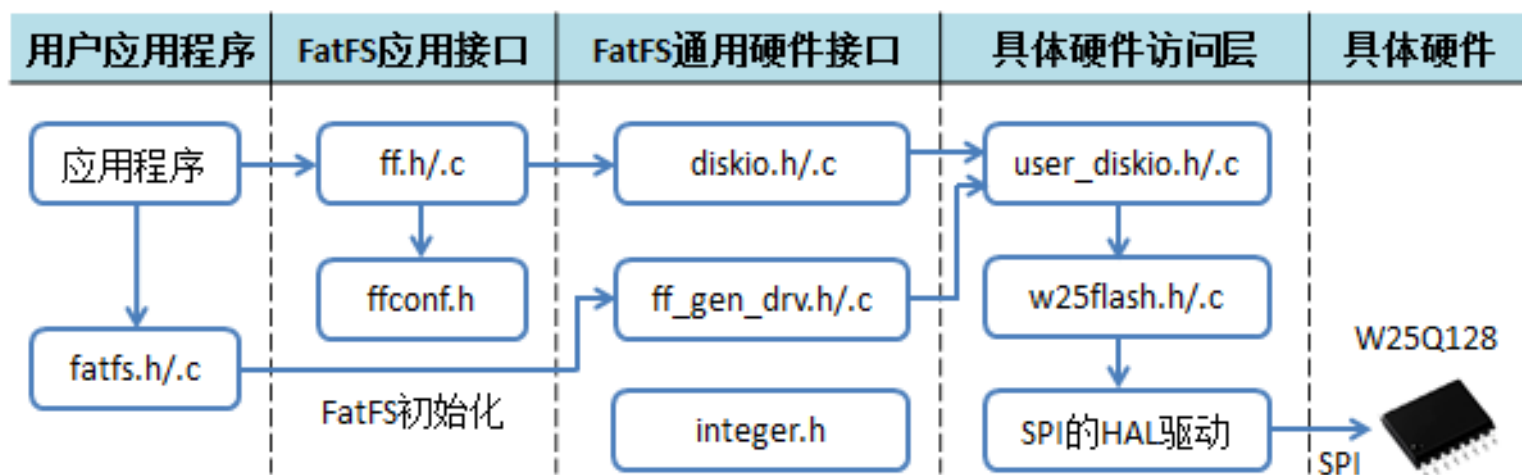
参数	默认值	可设置内容和功能描述
FS_TINY	Disabled	微小缓存区模式，如果设置为Enabled，每个文件对象(FIL)可减少内存占用512字节
FS_EXFAT	Disabled	是否支持exFAT文件系统，当_USE_LFN设置为0时，这个参数只能设置为Disabled
FS_NORTC	Dynamic timestamp	如果系统有RTC可以提供实时时间，就设置为Dynamic timestamp。如果系统没有RTC提供实时的时间，就选择为Fixed timestamp
FS_REENTRANT	Disabled	设置FatFS的可重入性，在CubeMX里没有启用FreeRTOS时，这个参数只能设置为Disabled，如果启用了FreeRTOS，这个参数只能设置为Enabled
FS_TIMEOUT	1000	超时设置，单位是节拍数。FS_REENTRANT设置为Disabled时，此参数无效
FS_LOCK	2	如果要启用文件锁定功能，设定FS_LOCK的值等于或大于1，表示可同时打开的文件个数。设定值范围0到255

12.4.4 项目中FatFS的文件组成

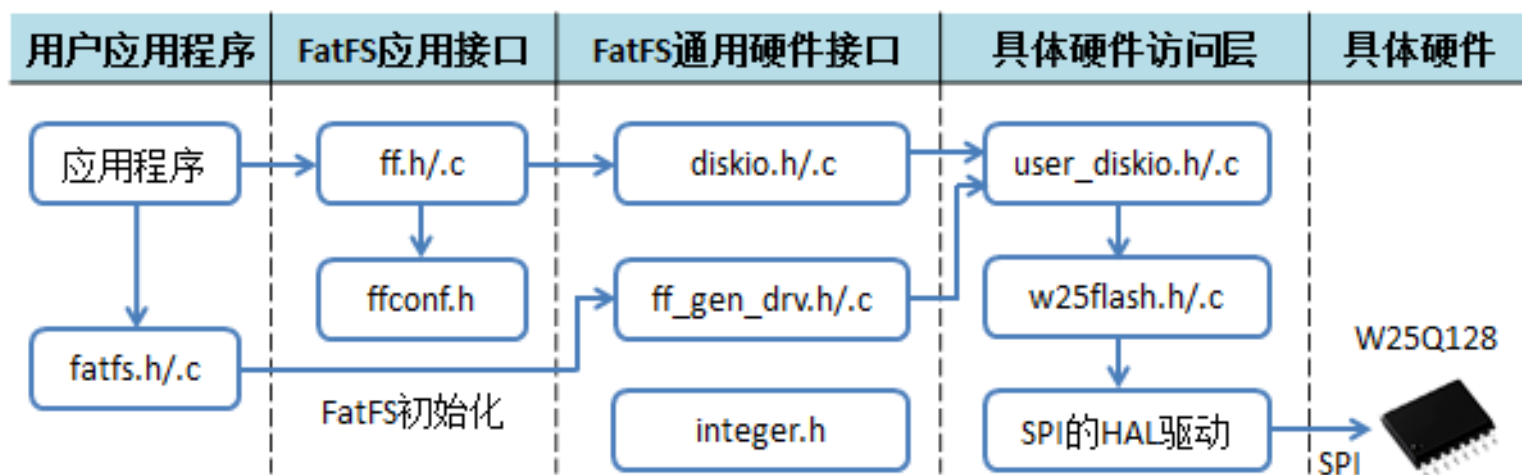
- `ff.h`和`ff.c`，FatFS应用程序接口API函数文件，与具体硬件无关
- `diskio.h`和`diskio.c`，存储介质Disk IO访问通用接口函数所在的文件
- `ff_gen_drv.h`和`ff_gen_drv.c`，实现驱动器列表管理功能的文件，FatFS初始化时就调用其中的函数
FATFS_LinkDriver()链接一个驱动器
- `\option\syscall.c`，这是在使用RTOS系统时需要实现的一些函数的代码，如果使用FreeRTOS，需要重新实现一些函数



Middlewares目录下FatFS的文件



- fatfs.h中包含FatFS初始化函数`MX_FATFS_Init()`，在主程序中进行外设初始化时会调用这个函数
- `MX_FATFS_Init()`会调用文件`ff_gen_drv.h`中的函数`FATFS_LinkDriver()`，将文件`user_diskio.h`中定义的驱动器对象`USER_Driver`链接到FatFS管理的驱动器列表里，相当于完成了驱动器的注册



- 文件user_diskio.c中实现了针对W25Q128芯片的Disk IO访问函数。文件user_diskio.h中定义了驱动器对象USER_Driver以及针对W25Q128芯片的Disk IO函数
- 在文件user_diskio.c中实现W25Q128的Disk IO访问函数时，需要用到W25Q128的驱动程序文件w25flash.h/.c，而这个驱动程序使用SPI接口的HAL驱动程序实现对W25Q128的访问

12.4.5 FatFS初始化过程

1. 主程序

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_FSMC_Init();
    MX_SPI1_Init();           //SPI1初始化
    MX_FATFS_Init();          //FatFS初始化
    MX_RTC_Init();            //RTC初始化
    /* Infinite loop */
    while (1)
    {
    }
}
```

2. 文件fatfs.h和fatfs.c

文件fatfs.h中定义了FatFS初始化函数MX_FATFS_Init(), 还定义了几个变量。称这个存储介质为**USER逻辑驱动器**。文件fatfs.h的完整代码如下:

```
/* 文件: fatfs.h, 用户的FatFS定义文件----- */
#include "ff.h"
#include "ff_gen_drv.h"
#include "user_diskio.h"           // 定义了 USER_Driver 底层访问函数的文件

/* 下面的几个变量是在文件fatfs.c中定义的, 用extern定义, 向外公开这几个变量 */
extern uint8_t retUSER;           // 函数返回值
extern char USERPath[4];         // USER逻辑驱动器路径, 如"0:"
extern FATFS USERFatFS;          // 用于USER逻辑驱动器的文件系统对象
extern FIL USERFile;             // USER的文件对象

void MX_FATFS_Init(void);         // FatFS初始化函数
```

源程序文件fatfs.c的完整代码如下：

```
#include "fatfs.h"

uint8_t retUSER;           // 函数返回值
char    USERPath[4];      // USER逻辑驱动器路径，如"0:/"
FATFS   USERFatFS;        // 用于USER逻辑驱动器的文件系统对象
FIL     USERFile;         // USER的文件对象

void MX_FATFS_Init(void)
{
    /* FatFS: 连接 USER。 函数FATFS_LinkDriver()在文件ff_gen_drv.h中定义
    USER_Driver是在文件user_diskio.c中定义的一个Diskio_drvTypeDef结构体类型变量。
    执行这行代码的作用是将USER_Driver链接到FatFS管理的驱动器列表，将USERPath赋值为 "0:/" */
    retUSER = FATFS_LinkDriver(&USER_Driver, USERPath);
}

/* 一个Disk IO函数，用于读取RTC的时间，为FatFS提供时间戳数据 */
DWORD get_fattime(void)
{
    /* USER CODE BEGIN get_fattime */
    return 0;
    /* USER CODE END get_fattime */
}
```

3. 文件user_diskio.h和user_diskio.c

文件user_diskio.h中只有一行有效语句，就是声明了变量USER_Driver

```
extern Diskio_drvTypeDef USER_Driver;
```

变量USER_Driver是在文件user_diskio.c中定义的，这个文件里包含SPI-Flash芯片的Disk IO函数框架，需要自己编写代码实现这些函数。

文件user_diskio.c的完整代码如下

```
/* 文件: user_diskio.c -----*/  
#include <string.h>  
#include "ff_gen_drv.h"  
  
static volatile DSTATUS Stat = STA_NOINIT; /* Disk status */  
  
/* 用户器件的Disk IO函数 -----*/  
DSTATUS USER_initialize (BYTE pdrv);  
DSTATUS USER_status (BYTE pdrv);  
DRESULT USER_read (BYTE pdrv, BYTE *buff, DWORD sector, UINT count);  
DRESULT USER_write (BYTE pdrv, const BYTE *buff, DWORD sector, UINT  
    count);  
DRESULT USER_ioctl (BYTE pdrv, BYTE cmd, void *buff);
```

```
/* 下面的代码是对结构体变量USER_Driver的各成员变量赋值，  
结构体类型Diskio_drvTypeDef在文件 ff_gen_drv.h中定义  
其成员变量都是函数指针，赋值使其指向本文件中定义的USER_ 函数 */  
Diskio_drvTypeDef USER_Driver =  
{  
    USER_initialize,      // 函数指针 disk_initialize 指向函数USER_initialize()  
    USER_status,          // 函数指针 disk_status 指向函数USER_status()  
    USER_read,            // 函数指针 disk_read 指向函数USER_read()  
    USER_write,           // 函数指针 disk_write 指向函数USER_write()  
    USER_ioctl,           // 函数指针 disk_ioctl 指向函数USER_ioctl()  
};
```

结构体Diskio_drvTypeDef类型的变量USER_Driver各成员变量是函数指针，指向本文件中定义的“USER_”函数，即具体的DiskIO函数

然后是几个 “USER_” 函数的框架

```
/* Private functions -----*/
/* 初始化驱动器，pdrv是驱动器号，如0 */
DSTATUS USER_initialize (BYTE pdrv )
{
    Stat = STA_NOINIT;
    return Stat;
}

/* 获取Disk状态信息 */
DSTATUS USER_status (BYTE pdrv )
{
    Stat = STA_NOINIT;
    return Stat;
}

/* 从某个扇区开始，读取1个或多个扇区的数据到缓存区buff */
DRESULT USER_read (BYTE pdrv, BYTE *buff, DWORD sector,UINT count )
{
    return RES_OK;
}
```

```

/* 将缓存区buff里的数据写入1个或多个扇区 */
DRESULT USER_write (BYTE pdrv,const BYTE *buff,DWORD sector, UINT
    count )
{
    return RES_OK;
}

/* I/O控制操作 */
DRESULT USER_ioctl (BYTE pdrv, BYTE cmd, void *buff )
{
    DRESULT res = RES_ERROR;
    return res;
}

```

针对SPI-Flash芯片的FatFS移植主要就是在文件
user_diskio.c中完善这几个“USER_”函数，实现存储介
质初始化、读取介质状态信息、以扇区为基本单位数据读写

4. 文件ff_gen_drv.h和ff_gen_drv.c


文件ff_gen_drv.h中定义了单个驱动器的硬件访问接口结构体类型Diskio_drvTypeDef，还定义了驱动器组的硬件访问结构体类型Disk_drvTypeDef。

定义了函数FATFS_LinkDriver()，就是MX_FATFS_Init()内调用的函数，用于将文件user_diskio.h中定义的驱动器对象USER_Driver链接到FatFS管理的驱动器列表里。

文件ff_gen_drv.h的完整代码如下：

```
/* 文件： ff_gen_drv.h -----*/
#include "diskio.h"
#include "ff.h"
#include "stdint.h"

/* Exported types -----*/
/* 驱动器Disk IO访问结构体,定义了Disk IO访问的5个函数指针 */
typedef struct
{
    DSTATUS (*disk_initialize)    (BYTE);        // 初始化驱动器
    DSTATUS (*disk_status)       (BYTE);        // 获取磁盘状态
    DRESULT (*disk_read)         (BYTE, BYTE*, DWORD, UINT);    // 读取数据
    DRESULT (*disk_write)        (BYTE, const BYTE*, DWORD, UINT); // 写入数据
    DRESULT (*disk_ioctl)        (BYTE, BYTE, void*);    // I/O 控制操作
}Diskio_drvTypeDef;
```



文件ff_gen_drv.h的代码

```
/* 全局的驱动器组结构体定义，管理所有驱动器（卷），每个驱动器有自己的Disk IO函数 */
typedef struct
{
    uint8_t      is_initialized[_VOLUMES];    //物理驱动器是否初始化,数组
    const Diskio_drvTypeDef  *drv[_VOLUMES]; //物理驱动器IO接口，数组
    uint8_t      lun[_VOLUMES];               //物理驱动器号，数组
    volatile uint8_t nbr;                      //逻辑驱动器个数
}Disk_drvTypeDef;

uint8_t FATFS_LinkDriver(const Diskio_drvTypeDef *drv, char *path);
uint8_t FATFS_UnLinkDriver(char *path);
uint8_t FATFS_LinkDriverEx(const Diskio_drvTypeDef *drv, char *path, BYTE lun);
uint8_t FATFS_UnLinkDriverEx(char *path, BYTE lun);
uint8_t FATFS_GetAttachedDriversNbr(void);
```

这是驱动器组的定义，有3个数组型成员变量，
_VOLUMES是管理的驱动器个数FatFS可以管理多个驱动器，如同时使用SD卡和SPI-Flash

文件ff_gen_drv.c的代码如下：

```
/* 文件： ff_gen_drv.c -----*/
#include "ff_gen_drv.h"

Disk_drvTypeDef disk = {{0},{0},{0},0};           //全局变量， FatFS连接的驱动器列表

/* 连接一个diskio兼容的驱动器， 调用函数FATFS_LinkDriverEx()。
 * 参数drv: 一个驱动器的Disk IO驱动结构体
 * 参数path:逻辑驱动器路径名称
 * 返回值： 0表示成功， 1表示失败
 */
uint8_t FATFS_LinkDriver (const Diskio_drvTypeDef *drv, char *path)
{
    return FATFS_LinkDriverEx(drv, path, 0);
}
```


/* 连接一个diskio兼容的驱动器，将逻辑驱动器路径格式加1，最多连接10个驱动器。

* 参数drv: 一个驱动器的Disk IO驱动结构体

* 参数path: 逻辑驱动器路径

* 参数lun: 仅用于U盘，用于加入multi-lun 管理，其他介质此参数必须是0

* 返回值: 0表示成功， 1表示失败

*/

uint8_t FATFS_LinkDriverEx(const Diskio_drvTypeDef *drv, char *path, uint8_t lun)

{

uint8_t ret = 1;

uint8_t DiskNum = 0;

if(disk.nbr < _VOLUMES)

//当前驱动器个数小于设置的卷个数

{

disk.is_initialized[disk.nbr] = 0;

//驱动器是否已经初始化

disk.drv[disk.nbr] = drv;

//驱动器的Disk IO 驱动结构体

disk.lun[disk.nbr] = lun;

//用于U盘

DiskNum = disk.nbr++;

//赋值后加1

path[0] = DiskNum + '0';

//逻辑驱动器路径名，如"0:/"

path[1] = ':';

path[2] = '/';

path[3] = 0;

ret = 0;

}

return ret;

}

5. 文件diskio.h和diskio.c

文件diskio.h是Disk IO访问的通用定义文件，其中定义了基本的结构体、宏定义和Disk IO访问通用函数

```
/* 文件： diskio.h ----- */

/* Disk IO 访问函数原型 */
DSTATUS disk_initialize( BYTE pdrv);
DSTATUS disk_status( BYTE pdrv);
DRESULT disk_read( BYTE pdrv, BYTE* buff, DWORD sector, UINT count);
DRESULT disk_write( BYTE pdrv, const BYTE* buff, DWORD sector, UINT count);
DRESULT disk_ioctl( BYTE pdrv, BYTE cmd, void* buff);
DWORD get_fattime( void);
```

文件diskio.h中还有其他一些宏定义，表示指令码、参数类型等

源程序文件diskio.c的完整代码如下：

```
/* 文件: diskio.c, Disk IO访问通用程序文件 -----*/
extern Disk_drvTypeDef disk; //disk在文件ff_gen_drv.c中定义,是所有驱动器的管理变量

/* 获取磁盘状态, 参数pdrv:物理驱动器, 如0 */
DSTATUS disk_status (BYTE pdrv)
{
    DSTATUS stat;
    stat = disk.drv[pdrv]->disk_status(disk.lun[pdrv]); //执行驱动器的disk_status()
    return stat;
}

/* 初始化一个驱动器的硬件接口, 参数pdrv:物理驱动器, 如0 */
DSTATUS disk_initialize (BYTE pdrv )
{
    DSTATUS stat = RES_OK;
    if(disk.is_initialized[pdrv] == 0) //未初始化
    {
        disk.is_initialized[pdrv] = 1;
        stat = disk.drv[pdrv]->disk_initialize(disk.lun[pdrv]); //执行驱动器的初始化函数
    }
    return stat;
}
```

```

/* 读取扇区的数据 */
DRESULT disk_read (BYTE pdrv, // 物理驱动器号
    BYTE *buff, // 读出数据的缓存区
    DWORD sector, // 扇区地址
    UINT count // 需要读取的扇区的个数
)
{
    DRESULT res;
    //执行驱动器的读取扇区数据的函数disk_read()
    res = disk.drv[pdrv]->disk_read(disk.lun[pdrv], buff, sector, count);
    return res;
}

#if _USE_WRITE == 1
DRESULT disk_write (
    BYTE pdrv, // 物理驱动器号
    const BYTE *buff, // 需要写入的数据缓存区
    DWORD sector, // 扇区地址
    UINT count // 需要写入的扇区个数
)
{
    DRESULT res;
    //执行驱动器的数据写入函数disk_write()
    res = disk.drv[pdrv]->disk_write(disk.lun[pdrv], buff, sector, count);
    return res;
}
#endif /* _USE_WRITE == 1 */

```

本示例只有一个驱动器，disk.drv[pdrv]就是USER_Driver，也就是User-defined驱动器，它的Disk IO函数就是文件user_diskio.c中前缀为“USER_”的几个函数。

```

/* I/O 控制操作 */
#if _USE_IOCTL == 1
DRESULT disk_ioctl (
    BYTE pdrv,           // 物理驱动器号，如0
    BYTE cmd,            // 控制码，就是diskio.h中的一些宏定义常数
    void *buff           // 发送/接收控制数据的缓存区
)
{
    DRESULT res;
    //执行驱动器的IO控制函数disk_ioctl()
    res = disk_drv[pdrv]->disk_ioctl(disk.lun[pdrv], cmd, buff);
    return res;
}
#endif /* _USE_IOCTL == 1 */

```

```

/* 从RTC获取时间作为文件系统的时间戳 */

```

```

__weak DWORD get_fattime(void)

```

```

{
    return 0;
}

```

函数get_fattime()使用了编译修饰符__weak，在任何文件里都可以重新实现这个函数，在文件fatfs.c中就重新定义了这个函数的框架。

12.4.6 针对SPI-Flash芯片的Disk IO函数实现

要实现Flash芯片W25Q128的FatFS移植，只需实现文件user_diskio.c中前缀为“USER_”的几个Disk IO函数，以及文件fatfs.c中的函数get_fattime()。

1. 获取驱动器状态的函数USER_status()

```
//文件diskio.h中的驱动器状态位宏定义
#define STA_NOINIT      0x01    // 驱动器未初始化
#define STA_NODISK     0x02    // 驱动器中无存储介质
#define STA_PROTECT     0x04    // 写保护
```

函数USER_status()用于返回驱动器的状态，如果存在以上的状态，就将相应的状态位置1，否则返回0x00即可。对于开发板上的W25Q128芯片来说，没有写保护问题，只存在是否已初始化的问题。

```

/* 文件： user_diskio.c ----- */
/* USER CODE BEGIN DECL */
#include <string.h>
#include "ff_gen_drv.h"
#include "w25flash.h"          //开发板上的W25Q128芯片的驱动程序
static volatile DSTATUS Stat = STA_NOINIT;      //表示驱动器状态的私有变量
/* USER CODE END DECL */

DSTATUS USER_status( BYTE pdrv )
{
    /* USER CODE BEGIN STATUS */
    Stat = STA_NOINIT;          // 驱动器未初始化, Stat=0x01
    if (0 != Flash_ReadID()) // 读取Flash芯片的ID,只要不是0就表示已初始化
        Stat &= ~STA_NOINIT;    // Stat=0x00
    return Stat;
    /* USER CODE END STATUS */
}

```

输入参数pdrv是驱动器编号，本示例中只有一个驱动器，pdrv为0，也就无需区分驱动器。

2. 驱动器初始化函数USER_initialize()

函数USER_initialize()用于驱动器硬件接口的初始化，对于W25Q128来说，就是与其连接的SPI1接口的初始化。本示例中，SPI1接口的初始化是由CubeMX自动生成的函数MX_SPI1_Init()完成的，所以，这里无需再对SPI1接口初始化。

```
DSTATUS USER_initialize (BYTE pdrv )
{
    /* USER CODE BEGIN INIT */
    Stat =USER_status(pdrv); //获取驱动器状态
    return Stat;
    /* USER CODE END INIT */
}
```

这里调用函数USER_status()获取驱动器状态，而函数USER_status()的返回值总是0x00，所以表示初始化成功

3. 驱动器IO控制函数USER_ioctl()

函数USER_ioctl()用于执行Disk IO访问时的一些操作，如获取总的扇区个数，获取扇区大小等。

```
DRESULT USER_ioctl( BYTE pdrv, BYTE cmd, void *buff )
{
    DRESULT res = RES_OK;
    switch(cmd)
    {
        case CTRL_SYNC: // 完成挂起的写操作过程，在 _FS_READONLY == 0时用到
            break;

        case GET_SECTOR_COUNT: // 获取存储介质容量 (_USE_MKFS == 1 时需要)
            *(DWORD *)buff=FLASH_SECTOR_COUNT; // 总的扇区个数,4096个
            break;

        case GET_SECTOR_SIZE: // 获取扇区大小 (_MAX_SS != _MIN_SS 时需要)
            *(DWORD *)buff=FLASH_SECTOR_SIZE; // 每个扇区的大小，4096字节
            break;

        case GET_BLOCK_SIZE: // 获取擦除块的大小(_USE_MKFS == 1 时需要)
            *(DWORD *)buff=16; // W25Q128一个Block有16个扇区
            break;

    }
    return res;
}
```

4. 读取扇区数据的函数USER_read()

函数USER_read()用于从W25Q128读取一个或多个扇区的数据，完成后的代码如下：

```
DRESULT USER_read( BYTE pdrv, BYTE *buff, DWORD sector, UINT count)
{
    /* USER CODE BEGIN READ */
    uint32_t globalAddr= sector<<12; //扇区编号左移12位得绝对起始地址
    uint16_t byteCount = count<<12; //字节个数，左移12位就是乘以4096，每个扇区4096字节
    Flash_ReadBytes(globalAddr, buff, byteCount); //读取数据
    return RES_OK;
    /* USER CODE END READ */
}
```

其中，参数buff是用来存储读出数据的缓存区，sector是读取数据的起始扇区编号，count是要读出数据的扇区个数。

函数Flash_ReadBytes()是W25Q128驱动程序文件w25flash.h中定义的函数，用于读出数据，它需要数据绝对起始地址作为输入参数

```
//从任何地址开始读取指定长度的数据
//globalAddr: 开始读取的地址(24bit), pBuffer: 数据存储区指针, byteCount:
//要读取的字节数
void Flash_ReadBytes( uint32_t globalAddr, uint8_t* pBuffer, uint16_t
    byteCount)
{
    /* 详细代码略 */
}
```

W25Q128驱动程序函数实现原理详见《基础篇》第16章

5. 写数据到扇区的函数USER_write()

函数USER_write()用于将一个缓存区内的数据写入Flash芯片W25Q128，完成后的代码如下：

```
DRESULT USER_write (BYTE pdrv, const BYTE *buff, DWORD sector, UINT
    count)
{
    /* USER CODE BEGIN WRITE */
    uint32_t globalAddr = sector<<12;                //绝对地址
    uint16_t byteCount  = count<<12;                //字节个数
    Flash_WriteSector(globalAddr, buff, byteCount);
    return RES_OK;
    /* USER CODE END WRITE */
}
```

其中，buff是待写入Flash芯片的数据缓存区的指针，sector是起始扇区编号，count是需要写入的扇区个数。

6. 获取RTC时间的函数get_fattime()

函数get_fattime()用于获取RTC时间，作为创建文件或修改文件的时间戳数据。diskio.c中的函数get_fattime()是用编译修饰符__weak定义的弱函数，在文件fatfs.c中重新实现这个函数

```
DWORD get_fattime( void)
{
    /* USER CODE BEGIN get_fattime */
    RTC_TimeTypeDef sTime;
    RTC_DateTypeDef sDate;
    if (HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN) == HAL_OK)
    {
        HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);
        WORD date=(2000+sDate.Year-1980)<<9;
        date = date |(sDate.Month<<5) |sDate.Date;

        WORD time=sTime.Hours<<11;
        time = time | (sTime.Minutes<<5) | (sTime.Seconds>1);
        DWORD dt=(date<<16) | time;
        return dt;
    }
    else
        return 0;
    /* USER CODE END get_fattime */
}
```

函数get_fattime()需要返回一个DWORD类型的数，这个数的高16位是日期，低16位是时间

表12-3 结构体FILINFO中的fdate日期数据格式

数据位	数据范围	表示意义
15:9位	共7位，数据范围0-127	年份，实际年份是1980加上这个值，如39表示2019年
8:5位	共4位，有效范围1-12	月份，表示1到12月
4:0位	共5位，有效范围1-31	日期，表示1到31日

表12-4 结构体FILINFO中的ftime时间数据格式

数据位	数据范围	表示意义
15:11位	共5位，数据范围0-23	小时，表示0到23时
10:5位	共6位，有效范围0-59	分钟，表示0到59分
4:0位	共5位，有效范围0-29	秒/2，将这个值乘以2之后是秒，表示0到58秒

12.5 在SPI-Flash芯片上使用文件系统

12.5.1 主程序功能

12.5.2 磁盘格式化

12.5.3 获取FAT磁盘信息

12.5.4 扫描根目录下的文件和子目录

12.5.5 创建文件和目录

12.5.6 读取文本文件

12.5.7 读取二进制文件

12.5.8 获取文件信息

12.5.9 文件file_opear.h的完整定义

12.5.1主程序功能

完成硬件层移植后，就可以在W25Q128芯片上创建FAT文件系统，进行文件和目录的管理，以及文件读写操作。



```
FRESULT res= f_mount(&USERFatFS, "0:", 1); //挂载驱动器
```

主程序创建了2级菜单

第1级菜单

```
[1]KeyUp   =Format chip
[2]KeyLeft =FAT disk info
[3]KeyRight=List all entries
[4]KeyDown =Next menu page
```

第2级菜单

```
[5]KeyUp   =Write files
[6]KeyLeft =Read a TXT file
[7]KeyRight=Read a BIN file
[8]KeyDown =Get a file info
```


12.5.2 磁盘格式化

在主程序中响应菜单项 “[1]KeyUp =Format chip”，对Flash芯片进行格式化操作的代码如下：

```
BYTE workBuffer[FLASH_SECTOR_SIZE]; //FLASH_SECTOR_SIZE=4096
DWORD clusterSize=2*FLASH_SECTOR_SIZE; //cluster必须大于或等于1个扇区
FRESULT res=f_mkfs("0:", FM_FAT, clusterSize, workBuffer, FLASH_SECTOR_SIZE);
```

格式化过程需要一个工作缓存区数组workBuffer，其大小必须是扇区的整数倍。

系统中只有一个驱动器，SPI-Flash芯片的逻辑驱动器号是“0:”，格式化选项使用FM_FAT即可，FatFS会自动根据簇的个数决定文件系统类型。

12.5.3 获取FAT磁盘信息

菜单项 “[2]KeyLeft =FAT disk info” 是获取磁盘信息并在LCD上显示，响应代码就是调用测试函数fatTest_GetDiskInfo()

详见文件file_opear.c中这个函数的代码

调用函数f_getfree()获取磁盘剩余簇的个数，同时返回一个文件系统对象指针fs，这个FATFS结构体里还有分区类型、簇的大小、剩余簇个数、扇区大小、卷的条目个数等信息。

利用f_getfree()返回的这些参数就可以计算磁盘的信息，例如总的扇区个数、总的簇个数、剩余存储空间大小等。

获取磁盘信息在LCD上显示的数据如下：

FAT type=1

[1=FAT12,2=FAT16,3=FAT32,4=exFAT]

Sector size(bytes)=4096

Cluster size(sectors)=2

Total cluster count=2008

Total sector count=4016

Total space(KB)=16064

Free cluster count=2002

Free sector count=4004

Free space(KB)=16016

FAT文件系统管理数据的最小单位是簇，一个文件至少占用1个簇的空间。

当执行了菜单项[5]创建了4个文件和2个目录后，总共用了6个簇。这就是4个文件和2个目录都各自占用了1个簇的存储空间。

这里显示总的扇区个数是4016，而W25Q128总共有4096个扇区，因为FAT文件系统要占用一些扇区

12.5.4 扫描根目录下的文件和子目录

菜单项 “[3]KeyRight=List all entries” 用来扫描和显示根目录下的文件和子目录，响应代码调用测试函数 `fatTest_ScanDir()`

详见文件 `file_opear.c` 中这个函数的代码

- 扫描一个目录需要先用函数 `f_opendir()` 打开这个目录，然后用函数 `f_readdir()` 逐一读取目录下的项
- `f_readdir()` 读出项的信息是一个 `FILINFO` 结构体变量，包括文件名 `fname`、属性位 `fattrib` 等信息，当项的文件名为空时表示没有多的项可读了，通过属性位 `fattrib` 可以判断一个项是文件还是目录
- 目录扫描完之后需要用函数 `f_closedir()` 关闭目录

在主程序中调用这个函数时执行的是fatTest_ScanDir("0:/"), 所以会扫描根目录下的文件和目录。

执行操作后, 在LCD上显示如下的信息, 列出了4个文件和2个目录的名称。文件名和目录名自动用大写表示。

```
FILE README.TXT
```

```
FILE HELP.TXT
```

```
FILE ADC500.DAT
```

```
FILE ADC1000.DAT
```

```
DIR SUBDIR1
```

```
DIR MYDOCS
```

12.5.5 创建文件和目录

菜单项 “[5]KeyUp =Write files” 用于创建4个文件和2个文件夹，保存到根目录下。响应这个菜单项的代码如下：

```
fatTest_WriteTXTFile("readme.txt",2019,3,5);  
fatTest_WriteTXTFile("help.txt",2016,11,15);  
fatTest_WriteBinFile("ADC500.dat",20,500);  
fatTest_WriteBinFile("ADC1000.dat",50,1000);  
f_mkdir("0:/SubDir1");  
f_mkdir("0:/MyDocs");
```

函数fatTest_WriteTXTFile()的代码如下：

```
void fatTest_WriteTXTFile( TCHAR *filename,uint16_t year, uint8_t month, uint8_t day)
{
    FIL file;
    FRESULT res= f_open(&file, filename, FA_CREATE_ALWAYS | FA_WRITE);
    if(res == FR_OK)
    {
        TCHAR str[]="Line1: Hello, FatFS\n";          //字符串必须有换行符"\n"
        f_puts(str, &file);                          //不会写入结束符"\0"
        TCHAR str2[]="Line2: UPC, Qingdao\n";
        f_puts(str2, &file);
        f_printf(&file, "Line3: Date=%d-%d-%d\n",year,month,day);

        LCD_ShowStr(10,LCD_CurY+LCD_SP15,(uint8_t *)"Write file OK: ");
        LCD_ShowStr(LCD_CurX,LCD_CurY,filename);
    }
    else
        LCD_ShowStr(10,LCD_CurY+LCD_SP15,(uint8_t *)"Open file error");
    f_close(&file);
}
```

打开文件的模式参数是FA_CREATE_ALWAYS | FA_WRITE，表示总是创建文件并执行写操作

函数fatTest_WriteBinFile()是创建二进制文件，所谓二进制文件就是按照一定的格式和顺序写入数据的文件，读出时也必须按照相应的格式和顺序读出。

```
void fatTest_WriteBinFile( TCHAR *filename, uint32_t pointCount, uint32_t sampFreq)
{
    FIL file;
    FRESULT res= f_open(&file,filename, FA_CREATE_ALWAYS | FA_WRITE);
    if(res == FR_OK)
    {
        TCHAR headStr[]="ADC1-IN5\n";
        f_puts(headStr, &file);    //写入字符串数据,以"\n"结尾, 不带"\0"

        UINT bw=0; //实际写入字节数
        f_write(&file, &pointCount, sizeof(uint32_t), &bw); //数据点个数
        f_write(&file, &sampFreq, sizeof(uint32_t), &bw); //采样频率
        uint32_t value=1000;
        for(uint16_t i=0; i<pointCount; i++,value++)
            f_write(&file, &value, sizeof(uint32_t), &bw);

        LCD_ShowStr(10,LCD_CurY+LCD_SP15,(uint8_t *)"Write file OK: ");
        LCD_ShowStr(LCD_CurX,LCD_CurY,filename);
    }
    f_close(&file);
}
```


对Flash类型介质进行文件数据写入时，要注意以下问题：

(1) **FatFS管理文件数据的最小单位是簇**，即使文件实际数据只有10个字节，它也会占用1个簇的存储空间。

(2) 在使用f_puts()、f_write()等函数向文件写入数据时，这些函数并不会直接导致底层Disk IO函数disk_write()被调用，否则一个小文件也会占用很多扇区。**FatFS内部有缓存机制**，使用f_puts()、f_write()等函数向文件写入的数据会被缓存，然后再写入存储介质。Flash存储介质擦除操作的最小单位是扇区。

(3) 虽然FatFS的写入操作有缓存机制，但是在向Flash类型的存储介质写入文件数据时**尽量避免频繁使用f_write()写入小数据量**。

在文件数据较大时应该自定义缓存区，缓存区大小为扇区大小的整数倍，数据先写入缓存区，然后用函数f_write()向文件一次写入一个缓存区的数据，即1个或多个扇区的数据，这样可以极大地提高数据写入的效率。

在第18章将LCD屏幕截图保存为BMP图片存入SD卡时就使用了这样的方法，相比于直接向文件逐个数据点写入，这种方法速度提高了几百倍。

12.5.6 读取文本文件

函数fatTest_ReadTXTFile()用于测试读取一个文本文件的内容，并在LCD上显示，其代码如下：

```
void fatTest_ReadTXTFile(TCHAR *filename)
{
    LCD_ShowStr(10,LCD_CurY,(uint8_t *)"Reading TXT file: ");
    LCD_ShowStr(LCD_CurX,LCD_CurY,filename);
    FIL file;
    FRESULT res= f_open(&file,filename, FA_READ); //以只读方式打开文件
    if(res == FR_OK)
    {
        TCHAR str[100];
        LcdFRONT_COLOR=lcdColor_WHITE;
        while (!f_eof(&file))
        {
            f_gets(str, 100, &file); //读取1个字符串,自动加上结束符"\0"
            LCD_ShowStr(10,LCD_CurY+LCD_SP10, (uint8_t *)str);
        }
        LcdFRONT_COLOR=lcdColor_YELLOW;
    }
    else if (res==FR_NO_FILE)
        LCD_ShowStr(10,LCD_CurY+LCD_SP10,(uint8_t *)"File does not exist");
    else
        LCD_ShowStr(10,LCD_CurY+LCD_SP10,(uint8_t *)"f_open() error");
    f_close(&file);
}
```

12.5.7 读取二进制文件

菜单项 “[7]KeyRight=Read a BIN file” 用于读取一个二进制文件ADC500.dat，并显示读取到的采样频率、数据点个数等信息。菜单项的响应代码是

```
fatTest_ReadBinFile("ADC500.dat");
```

函数fatTest_ReadBinFile()代码见源程序文件

执行这个菜单项，就可以显示从文件ADC500.dat读出的采样频率和数据点个数等主要信息，LCD上显示内容如下：

```
Reading Bin file: ADC500.dat
```

```
ADC1-IN5
```

```
Sampling freq= 500
```

```
Point count= 20
```

12.5.8 获取文件信息

菜单项 “[8]KeyDown =Get a file info” 用于获取文件 ADC1000.dat 的一些信息，如文件大小，修改日期等。

```
fatTest_GetFileInfo("ADC1000.dat");
```

LCD上显示如下内容。这个文件实际大小218字节，但是在Flash芯片上占用1个簇的存储空间，即2个扇区共8192字节。所以，为避免存储空间浪费，可以将簇的大小设置为1个扇区

```
File info of: ADC1000.dat  
File size(bytes)= 218  
File attribute= 0x20  
File Name= ADC1000.DAT  
File Date= 2019-12-15  
File Time= 15:32:02
```

函数fatTest_GetFileInfo()代码见源程序文件

在文件file_opera.h中定义了一个函数fat_GetTimeStamp()用于将FILINFO结构体里的日期和时间转换为RTC格式的日期和时间，以便于显示。其定义如下：

```
void fat_GetTimeStamp( const FILINFO *fno, RTC_DateTypeDef *sDate,  
    RTC_TimeTypeDef *sTime);
```

实际测试中发现：FatFS自动保存的文件修改时间只精确到分钟，秒数据永远是2。也就是说，即使为FatFS提供时间戳数据的函数get_fattime()中获取了RTC精确的秒数据，但是用函数f_stat()读取出的文件信息中的时间信息只准确到分钟。

12.5.9 文件file_opear.h的完整定义

文件file_opear.h是用于FatFS文件读写功能测试的头文件，与具体的存储介质无关，在后面各章的示例中还会用到，也会添加一些新的函数。

文件file_opear.h的完整定义见源程序文件

函数fat_GetFatTimeFromRTC()用于从RTC获取时间作为FatFS的时间戳数据，是对Disk IO函数get_fattime()的封装

封装为函数后，在文件fatfs.c中实现Disk IO函数get_fattime()时，就可以直接调用这个函数了，示意代码如下：

```
#include "file_opera.h"

DWORD get_fattime(void)
{
    return fat_GetFatTimeFromRTC();
}
```


练习任务

1. 看教材，练习本章的示例。