一个轻量级的嵌入式文件系统

简介

RanFS是基于`C(C89)`编写的，提供`POSIX`兼容的文件操作API，轻量级的文件系统库。 `RanFS`目标是为嵌入式设备提供功能完善和高效的文件操作API，通过上层接口封装实现对多种文件系统的读写或读支持。 可运行在资源很少的单片机环境里,比如 8051, PIC, AVR, ARM, DSP, Z80, 78K 等等。

功能简介

目前支持 `fat12/16/32` 可读写文件系统 `ntfs` `ext2` `ext3` `ext4` 只读文件系统，支持大于512字节扇区的盘，支持超过2T的硬盘，支持 `多分区` `多文件`，支持 `长文件名` 打开、创建、删除，支持通过扇区偏移量快速打开文件，支持 `创建目录` 等等。

移植要求

只需提供设备扇区读写函数和时间获取函数，详见移植说明。

内存使用

通过宏控制，来选择使用全局变量，还是需要时使用 `malloc` 进行动态分配，如使用 `malloc`，那么同时打开的分区和文件个数，取决于可用内存。如果选择使用全局变量，那么需要进一步的配置支持的扇区大小 分区数 文件数等，来决定内存空间的使用。

代码空间

由于完整的库占用比较大的空间，所以提供宏对每个API进行开启和关闭，来实现功能和空间的取舍，详见API配置。

基础API

rf_init

文件系统初始化函数

```
1  rf_int_t rf_init(rf_void_t);
```

参数 无参数

返回值 等于0成功，小于0失败，使用 `rf_errno` 获取错误代码。

注意 在使用任何API前，必须调用该函数对文件系统进行初始化，且必须返回0，才可以进行接下来的文件操作等。

rf_exit

文件系统退出函数，关闭打开的所有分区和文件。

```
1  rf_int_t rf_exit(rf_void_t);
```

参数 无参数

返回值 等于0成功，小于0失败，使用 `rf_errno` 获取错误代码。

注意

- 将自动关闭所有打开的文件。
- 调用该函数后，所有文件相关api不能再使用，如果需要再次使用，需再次调用rf_init进行初始化。

rf_version

获取文件系统版本函数

```
rf_version_t* rf_version(rf_void_t);
typedef struct _rf_version_t
{
    rf_u8_t     major;
    rf_u8_t     minor;
    rf_u8_t     alter;
    rf_u32_t    build;
}rf_version_t;
```

参数 无参数

返回值 `rf_version_t` *类型的版本值。

rf_errno

获取API执行失败后的错误代码

```
rf_int_t *rf_get_errno(rf_void_t);
#define rf_errno (*rf_get_errno())
```

参数 无参数

返回值 最后一次API失败时的错误代码

注意 只有在API返回失败后，才可调用该函数获取到错误代码。

rf_strerror

通过错误码得到相应的字符串解释，便于理解错误原因。

```
rf_char_t* rf_strerror(rf_int_t errnum);
```

参数 `errnum` `rf_errno`返回的错误代码

返回值 错误代码对应的字符串解释。

rf_part

获取设备分区信息

```
rf_int_t rf_part(rf_char_t *name, rf_part_param_t *param);
struct rf_part_param
{
    rf_int_t    sector_size;            /* 扇区大小字节数 0为自动获取获取 */
    rf_int_t    count;                  /* 存放有效分区数 */
    rf_part_t   part[RF_MAX_DISK_PART]; /* 存放分区结构 */
};
typedef struct rf_part_param rf_part_param_t, *rf_part_param_ref_t;
```

参数 `name` 设备名 `param` 设备分区结构体指针

返回值 等于0成功，小于0失败，使用`rf_errno`获取错误代码。

注意 最大支持的分区数，由`RF_MAX_DISK_PART`宏定义。

使用例子

```
rf_int_t ret;
rf_part_param_t pp = { 0 };

/* 枚举设备的分区消息 */
ret = rf_part( "udisk", &pp );
if( ret < 0 )
{
    printf( "rf_mount err: [%s]\n", rf_strerror(rf_errno) );
    return 1;
}
for (i = 0; i < pp.count; i++)
    printf("part_offset=%lld size=%lld\n", pp.part[i].start,
pp.part[i].size);
```

rf_mount

挂载设备分区

```c
rf_int_t rf_mount(
    const rf_char_t *name,
    const rf_char_t *path,
    const rf_char_t *fs_type,
    rf_int_t flags,
    const rf_mount_ctx_t *ctx
);
struct rf_mount_ctx
{
    rf_int_t    sector_size;    /* 指定设备扇区大小字节 */
    rf_lloff_t  volume_offset;  /* 分区物理扇区绝对偏移 */
#if RF_OPT_HAVE_CODE_PAGE
    rf_char_t   *code_page;     /* fat ext2 等文件系统的代码页 exfat ntfs 不需要配置代码页 */
#endif
};
typedef struct rf_mount_ctx rf_mount_ctx_t, *rf_mount_ctx_ref_t;
```

参数 `name` 设备名 `path` 挂载盘符 从`a:`盘开始到`z:`盘，支持的最多挂载的分区数由`RF_OPT_MAX_VOLUME`宏定义 `fs_type` 指定分区的文件系统类型 目前支持 `fat` `ntfs` `ext` `flags` 挂载标志， `RF_O_RDONLY` 只读挂载。`RF_O_RDWR` 支持读写挂载。 `ctx` 挂载上下位参数见`rf_mount_ctx_t`结构体注释。

返回值 等于0成功，小于0失败，使用`rf_errno`获取错误代码。

注意

- 只有将设备分区挂载成功后，才可以进行接下来的文件等操作。
- 一个设备分区，可只读挂载到多个盘符上，但只能读写挂载到一个盘符上，否则可能将破坏文件系统!!

使用例子

```c
rf_int_t ret;
rf_mount_ctx_t mc = { 0 };

/* 演示对第一个分区进行操作 pp.part[0].start为rf_part获取的设备第一个分区偏移 */
mc.volume_offset = pp.part[0].start;
/* 扇区字节数 从底层自动获取 */
mc.sector_size = 0;
/* 挂载u盘到a:盘符 需要对设备的读写权限 假定该分区为fat文件系统 */
ret = rf_mount( "udisk", "a:", "fat", RF_O_RDWR, &mc);
if( ret < 0 )
  printf( "rf_mount err: [%s]\n", rf_strerror(rf_errno) );
```

rf_unmount

卸载设备分区

```c
rf_int_t rf_unmount(const rf_char_t *path);
```

参数 `path` 需要卸载的盘符路径

返回值 等于0成功，小于0失败，使用`rf_errno`获取错误代码。

注意

- 卸载成功后，分区上所有打开的文件，将被关闭
- 卸载成功后，不能再对该分区进行任何文件操作。

rf_statfs

获取文件系统相关的信息

```
rf_int_t rf_statfs(const rf_char_t *path, struct rf_statfs *buf);
struct rf_statfs
{
    rf_u16_t        f_type;                 /* 文件系统类型 见rfs.h 文件系统类型定义 */
    rf_u16_t        f_ssize;                /* 一个扇区的字节 */
    rf_u16_t        f_bsize;                /* 一个簇的字节数 */
    rf_u32_t        f_bfree;                /* 分区有多少个空余簇 */
    rf_u32_t        f_blocks;               /* 分区拥有的总簇数 */
};
typedef struct rf_statfs rf_statfs_t, *rf_statfs_ref_t;
```

参数 `path` 需查询文件系统所在盘符。 `buf` `struct rf_statfs` 结构体的指针变量，用于储存文件系统相关的信息。

返回值 等于0成功，小于0失败，使用`rf_errno`获取错误代码。

使用例子

```
rf_statfs_t st;
ret = rf_statfs( "a:", &st );
if( ret < 0 )
  printf( "rf_statfs err: [%s]\n", rf_strerror(rf_errno) );
```

rf_open

打开文件函数

```
rf_int_t rf_open(const rf_char_t *pathname, rf_int_t flags);
```

参数 `pathname` 文件名。 `flags` 打开文件模式，支持多个模式宏进行或操作。

返回值 大于等于0为有效的文件描述符，否则返回-1，使用`rf_errno`获取错误代码。

注意

- 需注意 文件读写权限、文件名编码、中文路径等问题。

使用例子

```
rf_int_t fd;

fd = rf_open( "a:\\1.txt", RF_O_RDONLY );
if( fd < 0 )
    printf( "rf_open err: [%s]\n", rf_strerror(rf_errno) );
```

rf_raw_open

通过扇区偏移量快速打开文件

```
rf_int_t rf_raw_open(const rf_char_t *path, rf_fentry_t *entry,
rf_int_t flags);
struct _rf_fentry
{
    rf_lloff_t  sector;     /* 分区里文件所在扇区的相对偏移 */
    rf_u16_t    offset;     /* 文件所在扇区偏移 */
};
typedef struct _rf_fentry rf_fentry_t, *rf_fentry_ref_t;
```

参数 `path` 文件所在盘符 `entry` 见 `rf_fentry_t`类型定义 `flags` 打开文件模式，支持多个模式宏进行或操作。

返回值 大于等于0为有效的文件描述符，否则返回-1，使用`rf_errno`获取错误代码。

注意

- 相对于`rf_open`，该函数可以快速的打开文件，从而不再需要路径名来打开文件，标准POSIX并不提供该函数，这里增加该函数主要是为了提高打开文件速度和使用尽量少的内存来寻址文件（保存`rf_fentry_t`相对于文件路径名可能更加节省内存，保存文件描述符，虽然占用少，但文件描述符对应的文件结构体却远大于`rf_fentry_t`的空间占用）。
- 由于`rf_fentry_t`是文件在分区里的相对扇区偏移量，所以只对当前操作的分区有效，不具备`rf_open`根据文件名，可以跨设备的特性。
- 如何获取文件的`rf_fentry_t`信息，可使用rf_stat和rf_readdir_i函数进行获取。

rf_close

关闭文件标识符

```
rf_int_t rf_close(rf_int_t fd);
```

参数 `fd` 文件标识符

返回值 0成功，否则返回-1，使用`rf_errno`获取错误代码。

注意 文件打开使用完成，如果不关闭标识符，`rf_open`最后可能无法再打开文件。

rf_fstat

获取文件相关信息

```
rf_int_t rf_fstat(rf_int_t fd, rf_stat_t *buf);
struct rf_stat
{
    rf_fentry_t     entry;          /* 文件所在扇区的偏移量 */
    rf_u32_t        attribute;      /* 文件属性 */
    rf_fsize_t      current_offset; /* 文件当前偏移 */
    rf_fsize_t      size;           /* 文件大小 */
};
typedef struct rf_stat rf_stat_t, *rf_stat_ref_t;
```

参数 `fd` 文件描述符 `buf` 指向rf_stat_t结构体指针

返回值 0成功，否则返回-1，使用`rf_errno`获取错误代码。

rf_lseek

设置文件当前读写偏移

```
rf_off_t rf_lseek(rf_int_t fd, rf_off_t offset, rf_int_t whence)
```

参数 `fd` 文件标识符 `offset` 偏移字节数 `whence` 有以下取值：

- `RF_SEEK_SET` 当前偏移设置为`offset`字节。
- `RF_SEEK_CUR` 当前偏移设置为当前位置加上`offset`字节。
- `RF_SEEK_END` 当前偏移设置为文件大小加上偏移字节大/小。
  `whence`和`offset`两个参数，共同决定文件文件描述符fd的当前偏移量。

返回值 大于等于0时，为文件的当前偏移量，也就是距离文件开头多少个字节，否则返回-1，使用`rf_errno`获取错误代码。

注意

- 每一个已打开的文件都有一个读写位置, 当打开文件时通常其读写位置是指向文件开头, 若是以附加的方式打开文件(如`RF_O_APPEND`), 则读写位置会指向文件尾. 当`rf_read`或`rf_write`时, 读写位置会随之增加,`rf_lseek`便是用来控制该文件的读写位置. 参数`fd`为已打开的文件描述词, 参数`offset`为根据参数`whence`来移动读写位置的位移数。
- `rf_lseek`返回的是`rf_off_t`类型，根据支持文件系统的类型，自动调整为32位或64位有符号数，当为32位有符号数，最大支持文件2G的偏移。而fat文件系统最大4G文件大小，导致不能更改超过2G的偏移。

使用例子

```c
rf_off_t offset;

offset = rf_lseek(fd, 0, SEEK_SET); /* 将读写位置移到文件开头 */
if( offset < 0 )
    printf( "rf_lseek err: [%s]\n", rf_strerror(rf_errno) );
offset = rf_lseek(fd, 0, SEEK_END); /* 将读写位置移到文件尾时 */
offset = rf_lseek(fd, 0, SEEK_CUR); /* offset为当前文件的偏移 */
```

rf_read

读文件指定数据长度到缓冲区

```c
rf_ssize_t rf_read(rf_int_t fd, rf_void_t *buf, rf_size_t count);
```

参数 `fd` 文件标识符 `buf` 文件数据存放缓冲区 `count` 需要读取的数据长度

返回值 大于等于0为读取到文件数据的长度，否则返回-1，使用`rf_errno`获取错误代码。

注意 `rf_read`从当前的文件偏移量开始读取文件数据，当返回的数据长度小于请求的长度时，文件的所有数据读取完成。

使用例子

```c
rf_int_t ret;
rf_ssize_t len;
rf_char_t buf[32];

/* 演示循环读取文件 直到文件数据读取完成 */
for(;;)
{
    len = rf_read(fd, (rf_void_t *)buf, sizeof(buf));
    if( len < 0 )
    {
        printf( "rf_read err: [%s]\n", rf_strerror(rf_errno) );
        break;
    }

    /* 小于读取的长度 表明文件读取完成 */
    if( len < sizeof(buf) ) break;
}
```

rf_wrtie

写数据到文件

```
rf_ssize_t rf_write(rf_int_t fd, const rf_void_t *buf, rf_size_t
count);
```

参数 `fd` 文件标识符 `buf` 写入文件数据的缓冲区指针 `count` 需要写入数据长度

返回值 大于等于0为实际写入字节数，否则返回-1，使用`rf_errno`获取错误代码。

注意 `rf_write`从文件的当前偏移处开始写入数据，当返回的数据长度小于请求的长度时，表明分区已经写满。

使用例子

```
rf_int_t ret
rf_ssize_t len;
rf_char_t buf[32];

memset( buf, 0x55, sizeof(buf) );
/* 在文件的当前偏移处 写入一段数据 len返回写入数据有效长度 */
len = rf_write(fd, (rf_void_t *)buf, sizeof(buf));
if( len < 0 )
    printf( "rf_write err: [%s]\n", rf_strerror(rf_errno) );
```

rf_fsync

更新文件长度

```
rf_int_t rf_fsync(rf_int_t fd);
```

参数 `fd` 文件标识符

返回值 大于等于0为实际写入长度，否则返回-1，使用`rf_errno`获取错误代码。

注意 `rf_write`写入文件数据，当超过之前的文件长度后，文件长度不会自动更新到设备，如果这时出现断电等意外，超过之前文件长度后的数据在电脑上将无法读取，为避免这种情况出现，应该在写入数据后，周期性的调用该函数。

使用例子

```c
rf_char_t buf[16];
for (;;)
{
    ret = rf_write(fd, buf, strlen(buf) );
    if( ret < 0 )
    {
        printf( "rf_write err: [%s]\n", rf_strerror(rf_errno) );
        break;
    }
    /*rf_write len 每次返回长度为写入长度 当磁盘空间满时 返回长度小于写入长度 */
    if (ret < (rf_int_t)strlen(buf))
    {
        printf("Disk space full error\n");
        break;
    }
    /* 长时间写文件 调用rf_sync更新文件长度 当然不必这么频繁 本例只是演示 */
    ret = rf_fsync( fd );
    if( ret < 0 )
    {
        printf( "rf_fsync err: [%s]\n", rf_strerror(rf_errno) );
        break;
    }
}
```

rf_ftruncate

裁剪文件长度

```c
rf_int_t rf_ftruncate(rf_int_t fd, rf_off_t length);
```

参数 `fd` 文件标识符 `length` 文件长度

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

注意 该函数可以减小或扩大文件长度，当扩大文件长度时，扩展的文件数据为原扇区存在的数据，这里和PC上实现对扩展的文件数据清零是有区别的。

使用例子

```
rf_int_t fd, ret;

fd = rf_open("1.txt", RF_O_CREAT|RF_O_RDWR|RF_O_TRUNC );
show_error_stop("rf_open", fd);

/* 创建一个大文件 用于接下来高速写文件数据 */
ret = rf_ftruncate( fd, 100*1024*1024 );
show_error_stop("rf_ftruncate", ret);

/* 更新文件长度 */
ret = rf_fsync(fd);
show_error_stop("rf_fsync", ret);

/*
调用rf_write写数据 写入速度主要取决于 每次写入字节数
并可实现和写入物理扇区基本一样的速度
*/
rf_close( fd );
```

rf_fdelete

通过文件描述符删除文件

```
rf_int_t rf_fdelete(rf_int_t fd);
```

参数 `fd` 文件标识符

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

注意 `fd`需要只写或读写权限，才可以删除该文件。删除文件后，还需要调用`rf_close`关闭文件描述符。

使用例子

```
fd = rf_open( "1.txt", RF_O_RDWR );
if( fd >= 0 )
{
    ret = rf_fdelete(fd);
    if( ret < 0 )
        printf( "rf_fdelete err: [%s]\n", rf_strerror(rf_errno) );
    rf_close(fd);
}
```

rf_opendir_i

打开目录

```
rf_int_t rf_opendir_i(const rf_char_t *name);
```

参数 `name` 指定要打开的目录

返回值 大于等于0为有效的文件描述符，否则返回-1，使用`rf_errno`获取错误代码。

注意

- 这里和`POSIX`实现有所区别，主要为兼容不支持malloc的嵌入式设备。
- 该函数要求打开的文件必须是目录，否则返回错误。

rf_readdir_i

枚举目录

```
#define rf_readdir_i(fd, ptr) rf_readdir_ex( fd, "*", ptr )
rf_int_t rf_readdir_ex(rf_int_t fd, const rf_char_t *pattern,
rf_dirent_t *ptr);
struct rf_dirent
{
    rf_fentry_t     entry;          /*文件所处扇区和偏移*/
    rf_u32_t        attribute;      /*文件属性*/
    rf_int_t        type;           /*文件名编码类型·ansi unicode utf8 */
    rf_int_t        len;            /*文件名缓冲区长度*/
    rf_char_t*      name;           /*文件名缓冲区地址*/
};
typedef struct rf_dirent rf_dirent_t, *rf_dirent_ref_t;
```

参数 `fd` 目录文件描述符。 `ptr` `rf_dirent_t`结构体指针。

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。当错误代码为`RF_ENDFILE`时，目录枚举完成。

注意

- `rf_readdir_ex`函数的`pattern`参数，支持通配符，需要开启`RF_OPT_USE_PATTERN`宏。
- 由`RF_OPT_READDIR_LONG_NAME`宏定义，决定是否返回长文件名或短文件名。
- 该函数和`POSIX`实现由区别，主要为兼容不支持malloc的嵌入式设备。

使用例子

```c
rf_int_t ret, fd;
rf_dirent_t dt;
/* 某些MCU堆栈 可能无法分配RF_MAX_FNAME长度的局部变量 可使用全局变量代替 */
rf_char_t buf[RF_MAX_FNAME];

fd = rf_opendir_i("testdir");
if( fd < 0 )
{
    printf( "rf_opendir_i err: [%s]\n", rf_strerror(rf_errno) );
    return 1;
}

dt.name = buf;
dt.len = sizeof(buf);
for (;; )
{
    /*
    或使用rf_readdir_ex的函数 用通配符参数，实现只对mp3后缀名的文件进行枚举。
    ret = rf_readdir_ex( fd, *.mp3, &dt );
    */
    ret = rf_readdir_i(fd, &dt);
    if (ret < 0)
    {
        if (RF_ENDFILE == rf_errno) break;
        printf( "rf_readdir_i err: [%s]\n", rf_strerror(rf_errno) );
    }
    /*
    这里假定文件名编码类型为GB2312 可以正常打印输出 实际使用中如对ntfs exfat ext2等文件系统
    需要根据文件名编码类型进行相应的转码 然后才能正确的显示。
    */
    printf("%s\n", dt.name);
}
rf_closedir_i(fd);
```

rf_closedir_i

关闭目录

```
#define rf_closedir_i rf_close
```

注意 详见`rf_close`，打开目录后是需要关闭的，不然存在文件描述符泄露，导致最后无法再打开文件或目录。

rf_mkdir

创建目录

```
rf_int_t rf_mkdir(const rf_char_t *path);
```

参数 `path` 指定要创建的目录

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

rf_rmdir

删除目录

```
#define  rf_rmdir rf_unlink
rf_int_t rf_unlink(const rf_char_t *pathname);
```

参数 `pathname` 指定要删除的目录

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

注意

- 如果目录已被打开，在开启权限检查的条件下，将无法删除，在关闭文件权限检查的条件下，删除已被打开的目录，可能出现无法预料的情况，甚至会破坏文件系统。
- 删除的目录，需要目录下没有文件或目录，删除非空目录将导致目录下文件无法访问，而导致其占用的空间无法释放。

rf_chdir

改变当前目录

```
rf_int_t rf_chdir( const rf_char_t *path );
```

参数 `path` 指定切换的当前目录。

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

注意 打开文件操作（包括`rf_open` `rf_opendir_i` `rf_mkdir` `rf_rmdir`）除非指定绝对路径，否则都从当前目录下开始查找文件。

使用例子

```
ret = rf_chdir( "a:\\testdir" );
if( ret < 0 )
{
    printf( "rf_chdir err: [%s]\n", rf_strerror(rf_errno) );
    return 1;
}
/* 打开相对路径 将打开a:\\testdir目录下的1.txt文件*/
rf_open( "1.txt" );

/*
绝对路径 将打开a:\\1.txt
rf_open( "a:\\1.txt" );
*/
```

rf_getcwd

获取当前目录

```
rf_int_t rf_getcwd( rf_char_t *buf, rf_size_t size );
```

参数 `buf` 存放当前目录的缓存区地址 `size` 缓冲区的大小

返回值 0为成功，否则返回-1，使用`rf_errno`获取错误代码。

rf_rename

重命名文件

```
rf_int_t rf_rename(const rf_char_t *oldname, const rf_char_t *newname);
```

参数 `oldname` 为旧文件名。 `newname` 为新文件名。

返回值 修改文件名成功则返回0，否则返回-1，使用`rf_errno`获取错误代码。

注意 重命名文件： 如果`newname`指定的文件存在，则会被删除。 如果`newname`与`oldname`不在一个目录下，则相当于移动文件。

重命名目录 如果`oldname`和`newname`都为目录，则重命名目录。 如果`newname`指定的目录存在且为非空目录，则需先将`newname`目录删除。 重命名目录时，`newname`不能包含`oldname`作为其路径前缀。例如，不能将/usr更名为/usr/foo/testdir，因为老名字（ /usr/foo）是新名字的路径前缀，因而不能将其删除。

rf_unlink

通过文件名删除文件

```
rf_int_t rf_unlink(const rf_char_t *pathname);
```

参数 `pathname` 需要删除的文件名.

返回值 成功则返回0，失败返回-1，使用`rf_errno`获取错误代码。

注意 删除文件时，需要文件没被打开，在开启权限检测的条件下，会返回错误，没有开启权限检查时，删除打开的文件，可能会破坏文件系统!

一个简单的例子

```
#include "rfs.h"

rf_void_t show_error_stop(rf_char_t* info, rf_int_t ret)
{
    if (ret < 0)
    {
        printf("%s error [%s]\n", info, rf_strerror(rf_errno));
        while( 1 );
    }
}

rf_int_t main()
{
    rf_int_t ret, fd, len;
    rf_char_t buf[16];
    rf_mount_ctx_t mc = { 0 };
    rf_part_param_t pp = { 0 };
    rf_statfs_t st;

    /* 必须调用该函数 进行初始化 */
    ret = rf_init();
    show_error_stop( "rf_init", ret );

    /* 枚举设备的分区消息 */
    ret = rf_part( "udisk", &pp );
    show_error_stop("rf_part", ret);
    for (i = 0; i < pp.count; i++)
        printf("part_offset=%lld size=%lld\n", pp.part[i].start,
pp.part[i].size);

    /* 演示对第一个分区进行操作 */
    mc.volume_offset = pp.part[0].start;
    /* 扇区字节数 从底层自动获取 */
    mc.sector_size = 0;
    /* 挂载u盘到a: 需要对设备的读写权限 假定该分区为fat文件系统 */
    ret = rf_mount( "udisk", "a:", "fat", RF_O_RDWR, &mc);
```

```
    show_error_stop( "rf_mount", ret );

    /* 获取分区文件系统消息 */
    ret = rf_statfs( "a:", &st );
    show_error_stop( "rf_statfs", ret );
    printf( "Bytes per Sector:\t%d\n", st.f_ssize );
    printf( "Bytes per cluster:\t%d\n", st.f_bsize );
    printf( "Free clusters:\t\t%d\n", st.f_bfree );
    printf( "Total clusters:\t\t%d\n", st.f_blocks );

    /* 打开文件 返回文件句柄 */
    fd = rf_open( "a:\\1.txt", RF_O_RDONLY );
    show_error_stop( "rf_open", fd );

    /* 演示循环读取文件 直到文件数据读取完成 */
    for(;;)
    {
        len = rf_read(fd, (rf_void_t *)buf, sizeof(buf));
        show_error_stop( "rf_read", len );

        /* 小于读取的长度 表明文件读取完成 */
        if( len < sizeof(buf) ) break;
    }

    /* 关闭文件句柄 */
    ret = rf_close(fd);
    show_error_stop("rf_close", ret);

    /* 卸载挂载 关闭打开存储设备 */
    ret = rf_unmount("a:");
    show_error_stop("rf_umount", ret);

    /* 退出文件系统 */
    ret = rf_exit();
    show_error_stop("rf_exit", ret);
}
```

移植说明

移植rf_driver.c

rf_disk_open

打开挂载的设备，返回设备句柄。

```
rf_int_t rf_disk_open(const rf_char_t *device_name, rf_int_t flags,
rf_void_t* ctx, rf_handle_t* hdev);
```

参数

`device_name` [in] `rf_mount` 挂载的设备名 `flags` [in] `RF_O_RDONLY`只读挂载，`RF_O_RDWR` 读写挂载 `ctx` [in] 打开设备上下文参数，为rf_mount的ctx参数 `hdev` [out] 返回设备名，对应的句柄

返回值 `RF_ESUCCESS` 成功返回 `RF_ENOTRDY` 磁盘没有准备好或插入 `RF_EIO` 磁盘读写错误 `RF_EINVAL` 无效参数 `RF_ERROR` 未知错误

rf_disk_close

关闭设备句柄

```
rf_int_t rf_disk_close(rf_handle_t hdev)
```

参数 `hdev` [in] `rf_disk_open`输出的设备句柄

返回值 `RF_ESUCCESS` 成功返回 `RF_EINVAL` 无效参数 `RF_ERROR` 未知错误

rf_disk_ioctl

设备io控制函数

```
rf_int_t rf_disk_ioctl(rf_handle_t hdev, rf_int_t cmd, rf_void_t* buf,
rf_int_t len)
```

参数 `hdev` [in] 为`rf_disk_open` 返回的句柄 `cmd` [in] 支持如下命令

```
#define RF_CMD_GET_SECTOR_COUNT     1     /* 获取分区大小 */
#define RF_CMD_GET_SECTOR_SIZE      2     /* 获取设备扇区字节大小 */
#define RF_CMD_SET_SECTOR_SIZE      3     /* 设置设备扇区字节大小 */
#define RF_CMD_GET_BLOCK_SIZE       4     /* 获取擦除块大小 格式化函数使用 */
#define RF_CMD_CTRL_ERASE_SECTOR    5     /* 擦除设备块 格式化函数使用 */
#define RF_CMD_CTRL_SYNC            6     /* 刷新数据到磁盘 */
```

`buf` [in out] 输入或输出数据存放缓冲区 `len` [in] buf缓冲区长度

返回值 `RF_ESUCCESS` 成功返回 `RF_ENOTRDY` 磁盘没有准备好或插入 `RF_EIO` 磁盘IO错误 `RF_EINVAL` 无效参数 `RF_ERROR` 未知错误

rf_disk_read

磁盘设备物理扇区读函数

```
rf_int_t rf_disk_read(rf_handle_t hdev, rf_lloff_t sec_offset, rf_u32_t sec_count, rf_void_t *buf)
```

参数 `hdev` [in] 为 `rf_disk_open` 返回的句柄 `sec_offset` [in] 为扇区偏移 `sec_count` [in] 为读取扇区数 `buf` [out] 为扇区数据存放缓冲区

返回值 `RF_ESUCCESS` 成功返回 `RF_ENOTRDY` 磁盘没有准备好或插入 `RF_EINVAL` 无效参数 `RF_EIO` 磁盘IO错误 `RF_ERROR` 未知错误

注意 `sec_offset` 为当前分区相对偏移量，需要加上rf_disk_open参数 `param->ctx->volume_offset` 后转换为绝对偏移。

rf_disk_write

物理磁盘扇区写函数

```
rf_int_t rf_disk_write(rf_handle_t hdev, rf_lloff_t sec_offset,rf_u32_t sec_count,rf_void_t *buf);
```

参数 `hdev` [in] 为 `rf_disk_open` 返回的句柄 `sec_offset` [in] 为扇区偏移 `sec_count` [in] 为读取扇区数 `buf` [in] 为扇区数据存放缓冲区

返回值 `RF_ESUCCESS` 成功返回 `RF_EWRPRT` 磁盘写保护 `RF_ENOTRDY` 磁盘没有准备好或插入 `RF_EIO` 磁盘IO错误 `RF_EINVAL` 无效参数 `RF_ERROR` 未知错误

注意 `sec_offset`为当前分区相对偏移量，需要加上`rf_disk_open`参数param->ctx->volume_offset后转换为绝对偏移。

rf_getlocaltime

获取本地日期和时间

```
rf_int_t rf_getlocaltime( rf_tm_ref_t tm );

struct _rf_tm_t
{
    rf_u16_t    tm_year;    /* 年份 [1601 - 30827] */
    rf_u16_t    tm_mon;     /* 月份 取值区间为[1,12] */
    rf_u16_t    tm_mday;    /* 一个月中的日期 - 取值区间为[1,31] */
    rf_u16_t    tm_hour;    /* 时 - 取值区间为[0,23] */
    rf_u16_t    tm_min;     /* 分 - 取值区间为[0,59] */
    rf_u16_t    tm_sec;     /* 秒 –取值区间为[0,59] */
};
typedef struct _rf_tm_t rf_tm_t, *rf_tm_ref_t;
```

参数 `tm` [in] 本地时间返回结构体

返回值 `RF_ESUCCESS` 成功返回 `RF_ERROR` 不支持本地时间获取

注意 fat文件系统使用的是本地时间，作为文件的创建时间，最后更新时间。

移植rf_arch.c

rf_assert_handler

断言处理函数

```
rf_void_t rf_assert_handler(const rf_char_t * file, rf_int_t line);
```

参数 `file` [in] 断言发生的文件名 `line` [in] 断言发生的行号

注意 开始宏`RF_DEBUG`时，将使用该函数处理断言。

配置说明

所有配置在文件`rf_config.h`

设置CPU大小端

```
/*  1大端的CPU, 0为小端CPU 需要配置正确 否则调试版本下 会报断言错误 */
#define     RF_OPT_CPU_BIG_ENDIAN           0
```

是否使用malloc动态分配内存

```
/*
1 使用malloc动态分配内存 打开的分区、目录、文件等
0 使用全局变量打开的分区、目录、文件，最大数取决于相应文件系统配置宏定义
*/
#define     RF_OPT_USE_MALLOC               1
```

是否支持超过2T的设备

```
/*
1 支持超过4G个扇区个扇区的存储设备 分区数超过 4*1024*1024*1024 = 4G个
用于exfat ntfs ext2等文件系统 支持大容量的存储设备
*/
#define     RF_OPT_GT_4G_SECTORS            1
```

支持的分区格式

```
/* 1 支持MBR分区格式 0不支持 */
#define     RF_OPT_USE_MBR                  1
/* 1 支持GUID分区格式 0不支持 */
#define     RF_OPT_USE_GUID                 1
```

文件系统是否只读

```
/* 1 文件系统为只读 0 可读写文件系统 */
#define     RF_OPT_FS_READONLY              0
```

开启只读后，所有文件系统只能读文件，写文件相关函数将不再编译。

是否支持相对路径

```
/* 1支持打开相对路径 0不支持 */
#define     RF_OPT_FS_RPATH                 1
```

代码页配置

```
/* ----------------------------- 代码页设置-----------------------------
- */
/* 1 开启代码页支持 0 不支持代码页 */
#define     RF_OPT_HAVE_CODE_PAGE           1
#if RF_OPT_HAVE_CODE_PAGE
    /* 定义默认代码页为中文 */
    #define     RF_OPT_DEF_CODE_PAGE            RF_CODE_PAGE_936
    /* 定义支持的代码页 1为支持 0为不支持 */
    #define     RF_OPT_CODE_PAGE_437        1
    #define     RF_OPT_CODE_PAGE_720        1
    #define     RF_OPT_CODE_PAGE_737        1
    #define     RF_OPT_CODE_PAGE_775        1
    #define     RF_OPT_CODE_PAGE_850        1
    #define     RF_OPT_CODE_PAGE_852        1
    #define     RF_OPT_CODE_PAGE_855        1
    #define     RF_OPT_CODE_PAGE_857        1
    #define     RF_OPT_CODE_PAGE_862        1
    #define     RF_OPT_CODE_PAGE_866        1
    #define     RF_OPT_CODE_PAGE_932        1
    #define     RF_OPT_CODE_PAGE_936        1
    #define     RF_OPT_CODE_PAGE_949        1
    #define     RF_OPT_CODE_PAGE_950        1
#endif
```

文件名当前代码页

```
/* 打开文件名编码 */
#define     RF_OPT_PATH_CODE_PAGE           RF_CODE_PAGE_936
```

文件名缓冲区模式

```
/*
打开路径时 文件名需要临时缓冲区 用于对比等操作
可以使用全局变量 堆栈变量 malloc动态分配 不分配
1 使用全局变量
2 使用堆栈变量
3 malloc动态分配
4 不分配空间 rf_open 打开根目录后 rf_readir rf_raw_open来操作文件
*/
#define     RF_OPT_PATH_LOCAL                1
```

由于`fat` `ntfs`等文件系统，长文件名长度可达到512字节，需要根据环境选择这块内存如何分配。

文件名是否支持通配符

```
/* 1 支持通配符 "* ? "匹配  0不支持 */
#define     RF_OPT_USE_PATTERN              1
```

最大支持挂载分区数

```
/* 最多可挂载分区数 1到26 从a:到z: */
#define     RF_OPT_MAX_VOLUME                1
```

最大支持打开文件数

```
/* 不支持动态分配内存 使用静态文件句柄 */
#if RF_OPT_USE_MALLOC == 0
    /* 定义同时打开文件最大数 */
    #define     RF_OPT_MAX_FILE              1
#endif
```

最大支持设备扇区字节数

```
#if RF_OPT_USE_MALLOC == 0
    /* 定义支持磁盘扇区最大字节数 512 1024 2048 4096 */
    #define     RF_OPT_MAX_SECTOR_SIZE       512
#endif
```

文件是否支持权限检查

```
/* 多文件读写权限检测 相同文件支持多个同时读 只支持一个写 */
#define     RF_OPT_FILE_ACCESS_CHECK        1
```

rf_readdir_i 长文件名或短文件名

```
/* readdir 读取长文件名 或者 短文件名 fat ntfs
1 rf_readdir_i 返回长文件名
0 rf_readdir_i 返回短文件名
*/
#define     RF_OPT_READDIR_LONG_NAME        1
```

FAT文件系统配置

```
/* 1支持fat文件系统 0不支持 */
#define     RF_OPT_USE_FAT                  1

#if RF_OPT_USE_FAT
    /* 1支持fat12 0不支持  */
    #define     RF_OPT_USE_FAT12            1
    /* 1支持fat16 0不支持  */
    #define     RF_OPT_USE_FAT16            1
    /* 1支持fat32 0不支持 */
    #define     RF_OPT_USE_FAT32            1
    /* fat使用长文件名 */
    #define     RF_OPT_FAT_LONG_NAME        1
    /* 不支持动态内存分配 */
    #if RF_OPT_USE_MALLOC == 0
        /* 定义支持 最多支持打开分区数 */
        #define     RF_OPT_FAT_MAX_VOLUME   1
        /* 定义支持 最多可以打开文件个数 */
        #define     RF_OPT_FAT_MAX_FILE     1
    #endif
#endif
```

NTFS文件系统配置

```
/* 1支持ntfs文件系统 0不支持 */
#define     RF_OPT_USE_NTFS                    1

#if RF_OPT_USE_NTFS
    #if RF_OPT_USE_MALLOC == 0
        /* ntfs 最多支持打开分区数 */
        #define     RF_OPT_NTFS_MAX_VOLUME   1
        /* ntfs 最多可以打开文件个数 */
        #define     RF_OPT_NTFS_MAX_FILE     1
    #endif
#endif
```

EXT2/3/4文件系统配置

```
/* 1支持ext2/3/4 0不支持 */
#define     RF_OPT_USE_EXT2                    1

#ifdef RF_OPT_USE_EXT2
    #if RF_OPT_USE_MALLOC == 0
        /* ext2/3/4 最多支持打开分区数 */
        #define     RF_OPT_EXT2_MAX_VOLUME   1
        /* ext2/3/4 最多可以打开文件个数 */
        #define     RF_OPT_EXT2_MAX_FILE     1
    #endif
#endif
```

API配置

```
/* ----------------------------- api--------------------------------
*/
/* 以下1 为使用函数 0 为不使用该函数 缩小程序占用空间 */
/* rf_version */
#define RF_OPT_USE_VERSION                    1
/* rf_exit */
#define RF_OPT_USE_rf_EXIT                   1
/* rf_get_volume_handle_count rf_get_file_handle_count */
#define RF_OPT_USE_GET_HANDLE_COUNT           1
/* rf_set_volume_ctx rf_get_volume_ctx */
#define RF_OPT_USE_VOLUME_CTX                 1
/* rf_set_file_ctx rf_get_file_ctx */
#define RF_OPT_USE_FILE_CTX                   1
/* rf_parts */
#define RF_OPT_USE_RF_PART                    1
```

```
/* rf_mkfs */
#define RF_OPT_USE_MKFS                     1
/* rf_statfs */
#define RF_OPT_USE_STATFS                   1
/* rf_get_label */
#define RF_OPT_USE_GET_LABEL                1
/* rf_set_label */
#define RF_OPT_USE_SET_LABEL                1
/* rf_strerror */
#define RF_OPT_USE_STRERROR                 1
/* rf_raw_open */
#define RF_OPT_USE_RAWOPEN                  1
/* rf_fstat */
#define RF_OPT_USE_GET_FSTAT                1
/* rf_get_ftime */
#define RF_OPT_USE_GET_FTIME                1
/* rf_get_ltime */
#define RF_OPT_USE_GET_LTIME                1
/* rf_get_name */
#define RF_OPT_USE_GET_FNAME                1
/* rf_get_long_name rf_set_long_name */
#define RF_OPT_USE_GET_SET_LONG_NAME        1
/* rf_lseek */
#define RF_OPT_USE_FSEEK                    1
/* rf_chdir */
#define RF_OPT_USE_CHDIR                    1
/* rf_getcwd */
#define RF_OPT_USE_GETCWD                   1
/* rf_readdir_i */
#define RF_OPT_USE_READDIR                  1
/* rf_mkdir */
#define RF_OPT_USE_MKDIR                    1
/* rf_set_attrib */
#define RF_OPT_USE_SET_ATTRIB               1
/* rf_set_ftime */
#define RF_OPT_USE_SET_FTIME                1
/* rf_set_ltime */
#define RF_OPT_USE_SET_LTIME                1
/* rf_set_name */
#define RF_OPT_USE_SET_NAME                 1
/* rf_rname */
#define RF_OPT_USE_RENAME                   1
/* rf_ftruncate */
#define RF_OPT_USE_FTRUNCATE                1
/* rf_unlink */
#define RF_OPT_USE_UNLINK                   1
/* rf_fdelete */
#define RF_OPT_USE_FDELETE                  1
/* rf_findfirst rf_findnext rf_findclose */
```

```
64 #define RF_OPT_USE_FINDAPI                  1
```

相关参考

文件名

文件名编码

`rf_open` `rf_mkdir` `rf_opendir_i` `rf_unlink`等，打开文件路径名，默认编码为`CP936`，如果需要支持`utf8` `ucs2`其它编码的文件名，需要配置`RF_OPT_PATH_CODE_PAGE`为相应的编码，同时带上相应的代码页支持。

文件名最大长度

由宏`RF_MAX_FNAME`定义，由于不同文件系统支持的文件名最大长度不太相同，如

- fat 短文件名最大长度： 8(文件名)+1(.字符)+3(后缀名)+1(\0结束字符) = 13字节
- ext 2/3/4文件名最大长度： 255(文件名) + 1(\0结束字符) = 256字节
- ntfs和fat长文件名最大长度： 255*2（文件名)+ 2(unicode 两个\0 结束) = 512字节

假如只支持打开fat短文件名`RF_MAX_FNAME`定义13字节就够了。

FAT长文件名和短文件名

长文件名判断条件，以下满足一个既是长文件名

- 文件主名里有大小写字母或后缀名里有大小写字母。
- 文件主名长度大于8个字节，或文件后缀名大于3个字节。

```
"abcd1234.txx"  /* 是短文件名 文件名全小写*/

"ABCD1234.txx"  /* 是短文件名 文件主名或扩展名 不存在大小字母 */
"abcd1234.TXX"  /* 是短文件名 文件主名或扩展名 不存在大小字母 */

"ABCd1234.txt"  /* 是长文件名 文件主名里有大小字母 */
"ABCD1234.txX"  /* 是长文件名 后缀名里有大小字母*/

"ABCD12345.txt" /* 是长文件名 文件主名长度大于8个字节 */
"ABCD1234.txta" /* 是长文件名 文件后缀名 大于3个字节 */
```

由于文件名默认编码**CP936**，如果需要支持中文长文件名的创建及打开操作，需要开启`RF_OPT_HAVE_CODE_PAGE`宏和`RF_OPT_CODE_PAGE_936`宏。注意开启中文代码页的条件下会多占用几百KB的空间。

在不开启代码页的条件，可以支持对字母、数字、(及其它ANSI字符)组成的长文件名的创建和打开，如果这时打开中文长文件名会返回`RF_ENAMEENCODING`错误。

文件路径

路径分隔符

支持`\`或`/`分隔符。

```
/* 也就是如下路径名都是支持的 */
"a:\\test\\dir\\1.txt"
"a:/test/dir/1.txt"
```

绝对路径和相对路径

绝对路径：是从盘符开始的路径，如`a:\test\dir\1.txt` **相对路径**：是从当前目录开始的路径，假如当前目录为`a:\test`要打开上述文件，只需输入`dir\1.txt`，严格的相对路径写法应为`.\dir\1.txt`其中，.表示当前路径，在通道情况下可以省略，只有在特殊的情况下不能省略。 假如当前路径为`a:\test2`要打开上述文件，可以用路径打开`..\test\dir\1.txt`

其中，`..`为父目录。当前路径如果为`a:\test2\dir`则可以用`..\..\test\dir\1.txt`打开上述文件

另外，还有一种不包含盘符的特殊绝对路径，形如`\test\dir\1.txt`无论当前路径是什么，会自动从当前盘的根目录开始打开文件。

路径最大长度

路径的最大长度 由`RF_MAX_PATH`宏定义，文件系统本身对支持的路径最大长度没有最大限制。

中文路径

中文路径名，keil等默认编码**GB2312**，由于fat长文件名和ntfs文件名等是按**UCS2**编码存放，所以这里需要开启对**CP936**代码页支持，这需要多占用几百KB的空间，所以对空间较小的MCU，没必要支持代码页，在不支持代码页的情况，fat可以使用英文长文件名打开和创建等操作。但如果需要打开中文路径，会返回`RF_ENAMEENCODING`错误。

文件打开标志

- `RF_O_RDONLY` 以只读方式打开文件
- `RF_O_WRONLY` 以只写方式打开文件
- `RF_O_RDWR` 以可读写方式打开文件. 上述三种旗标是互斥的, 不可同时使用, 但可与下列的旗标利用OR(|)运算符组合.
- `RF_O_CREAT` 若欲打开的文件不存在则自动建立该文件.
- `RF_O_TRUNC` 若文件存在并且以可写的方式打开时, 此旗标会令文件长度清为0, 而原来存于该文件的资料也会消失.
- `RF_O_APPEND` 当读写文件时会从文件尾开始移动, 也就是所写入的数据会以附加的方式加入到文件后面.
- `RF_O_DIRECTORY` 如果参数pathname 所指的文件并非为一目录, 则会令打开文件失败。
- `RF_O_SECBUF` 使用分区的扇区缓冲区，读或写文件，要求从打开文件后每次读取或写入一个扇区的数据，该标志通常用在内存资源极小(小于1k的)MCU，整扇区快速读取文件数据，使用此标志，需要开启配置`RF_OPT_FILE_SECBUF`宏。

文件读写权限

一个文件只读可以被打开多次，只写或读写只能打开一次，使用文件简单权限检查，便于嵌入式设备环境实施，同时也可以通过`RF_OPT_FILE_ACCESS_CHECK`宏进行关闭，在保证不出现一个文件被多次写打开的情况下。在没开启宏的条件下，通过多个不同文件描述符对同一个文件写数据将会破坏文件系统!!

文件描述符

- 文件描述符是无符号的整数，使用它来标识打开的文件。文件描述符与包括相关信息（如文件的打开模式、文件的位置类型等）的文件对象相关联，这些信息被称作文件的上下文。
- 文件描述符的有效范围，在不支持`RF_OPT_USE_MALLOC`，是 0 到 `RF_OPT_MAX_FILE`，在支持`RF_OPT_USE_MALLOC`情况下，是从0到可用2048。

什么是盘符

盘符是磁盘分区设备的标识符，使用英文字母加上一个冒号: 来标识，从`a:`开始。最大支持的盘符数由`RF_OPT_MAX_VOLUME`宏定义，最大支持到`z:`。

支持的分区格式

- 支持MBR分区
- 支持GUID分区
- 没有分区表，直接一个分区的设备

错误代码

```
enum {
    RF_ESUCCESS,            /* 函数成功返回 非零为错误 */
    RF_ERROR,               /* 未知错误 在意外情况下出现 */
    RF_EINVAL,              /* 无效的参数 */
    RF_EBADF,               /* 错误的文件描述符*/
    RF_ENOMEM,              /* 分配内存失败，在支持动态分配内存情况下 */
    RF_EBUFSMALL,           /* 缓存区过小 */
    RF_EACCES,              /* 权限拒绝 */
    RF_EPERM,               /* 操作不支持 检查是否开启相应的宏支持 */
    RF_ENAMETOOLONG,        /* 文件名太长 */
    RF_ENAMEENCODING,       /* 文件名编码转换失败 检查开启相应的代码页宏支持 */
    RF_EISDIR,              /* 当要打开文件时 打开的是目录 返回该错误 */
    RF_ENOTDIR,             /* 当要打开目录时 打开的是文件 返回该错误 */
    RF_EMFILE,              /* 分配文件对象失败 打开文件过多 */
    RF_ENOENT,              /* 不存在目录或文件 检查文件路径是否正确 */
    RF_ENDFILE,             /* 目录枚举完成 会返回该错误 */
    RF_ENOSPC,              /* 分区写满 不能再写入数据 */
    RF_EFBIG,               /* 文件太大 fat文件系统写入超过4G字节时出现 */
    RF_EXDEV,               /* 跨分区连接文件错误 */
    RF_EMVOL,               /* 没有可用盘符分配 */
    RF_ENOTRDY,             /* 磁盘没有准备好或插入 */
    RF_EIO,                 /* 存储设备底层IO错误 */
    RF_ENOFS,               /* 分区的文件系统不能识别 */
    RF_EWRPRT,              /* 存储设备写保护 不能写数据 */
    RF_EROFS                /* 设备用只读挂载 不支持写操作 */
}
```

相关讨论

为什么需要文件读写权限检查

一个文件被多次读打开，同时读文件数据不存在任何问题，不需要任何权限检查。但如果一个文件，被多次读并存在写的打开，这里需要引入同步机制，否则会带来一系列的同步问题。所以这里为了方便处理，引入了文件权限检查，规则为：**一个文件可以多次读打开，写或读写打开，只能打开一次。**

为什么需要打开多个文件

主要为处理以下情况:

- 同时读多个文件，如果只支持打开一个文件读数据，需来回打开文件和改变文件偏移，导致效率低下。
- 一个大文件快速读多个不同偏移的数据，如果只打开一次文件，存在来回改变文件偏移情况，在改变偏移比较大的情况下会出现比较耗时的问题。