littleFS 文件系统

> **够用的硬件**
> **能用的代码**
> **实用的教程**
> 屋脊雀工作室编撰 -20181220
> 官网：www.wujique.com
> github:https://github.com/wujique/stm32f407
> 资料下载：https://pan.baidu.com/s/12o0Vh4Tv4z_O8qh49JwLjg

littlefs本人并未在实际项目（指已经出货并经长期验证的产品）中用过，以下相关信息仅仅为本人测试的结果

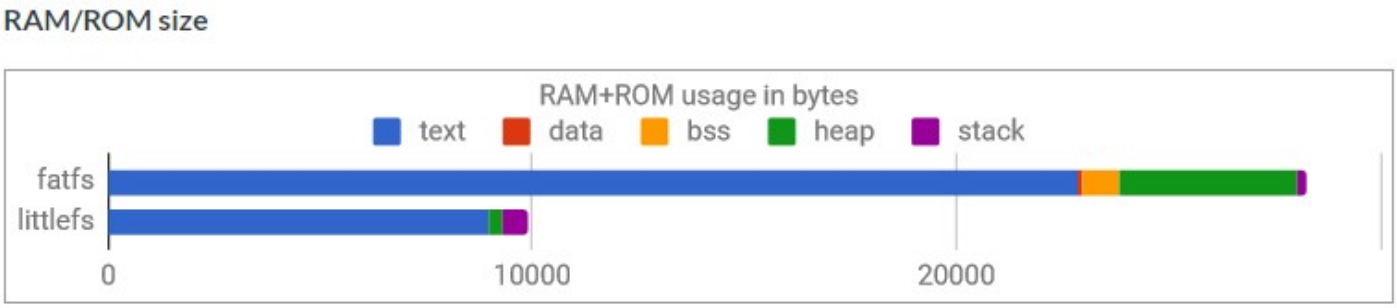
littlefs是一个用于FLASH的文件系统。
是ARM官方mbed项目的一个软件模块。
https://os.mbed.com/blog/entry/littlefs-high-integrity-embedded-fs/

概述

littlefs有3个特点：

1. RAM和FLASH用的少，也就是内存和代码和精简。
2. 掉电保护，设备突然断电，文件系统不会损坏。
3. 磨损均衡，因为FALSH的擦除/写次数有限制，通常是10万次。

与fatfs对比



磨损均衡对比

fatfs使用固定的块做文件系统管理，多次读写后，整个系统就坏了，但是其他块其实没用多少次。

本人并不建议大家在flash上使用fatfs

源码

源码在github上开源：
https://github.com/ARMmbed/littlefs

从github下载源码，解压，目录如下：

名称	修改日期	类型	大小
emubd	2018/11/16 15:45	文件夹	
tests	2018/11/16 15:45	文件夹	
.gitignore	2018/7/28 4:59	文本文档	1 KB
.travis.yml	2018/7/28 4:59	YML 文件	8 KB
DESIGN.md	2018/10/29 9:40	MD 文件	76 KB
lfs.c	2018/10/26 11:47	C 文件	70 KB
lfs.h	2018/10/26 11:20	H 文件	16 KB
lfs_util.c	2018/7/28 4:59	C 文件	1 KB
lfs_util.h	2018/7/28 4:59	H 文件	5 KB
LICENSE.md	2018/7/28 4:59	MD 文件	2 KB
Makefile	2018/7/28 4:59	文件	2 KB
README.md	2018/7/28 4:59	MD 文件	7 KB
SPEC.md	2018/10/24 17:40	MD 文件	21 KB

源码只有四个文件：

> lfs.c
> lfs.h
> lfs_util.c
> lfs_util.h

README.md文件中有一个例子教我们如何使用LFS。
SPEC.md是LFS概述，概要的说了LFS的设计理念。
DESIGN.md则是FLS设计细节说明。

移植使用

本次移植基于屋脊雀的STM32F407开发板，FLASH使用底板的FLASH，型号MX25L3206E。
相关的spi和spi flash驱动在此之前已经实现。

目前还不熟悉，最好根据官方说明移植，也就是参考 README.md。

添加应用文件

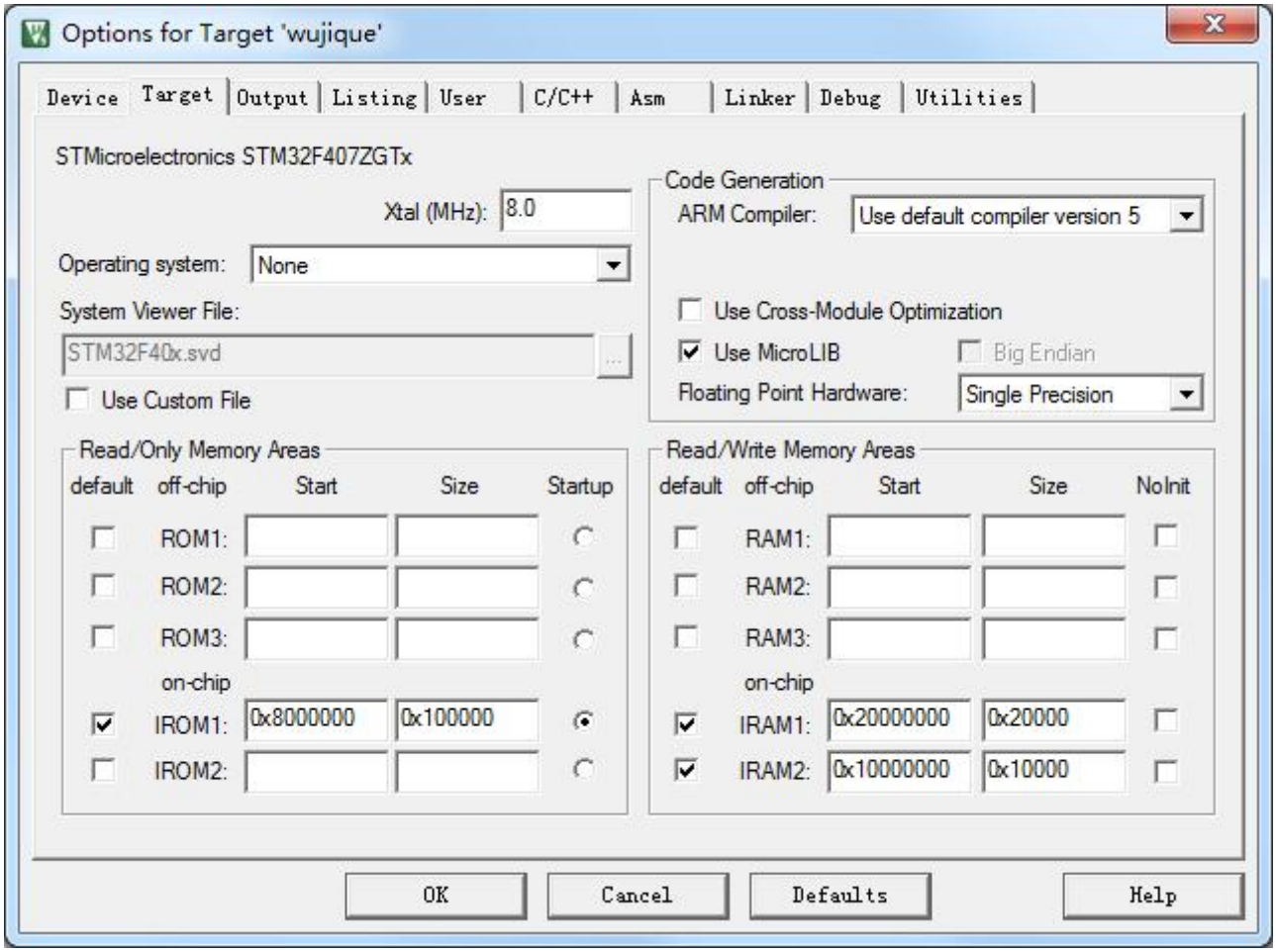
在应用中添加两个文件sys_littlefs.c、sys_littlefs.h，littlefs相关的配置和测试代码就放在这里。

添加lfs源码

将lfs四个源文件和刚刚创建的两个文件添加到工程，**直接编译**。
测试lfs源码是否能编译通过。

```
.\Objects\wujique.axf: Error: L6218E: Undefined symbol __aeabi_assert (referred from lfs.o).
Not enough information to list image symbols.
Not enough information to list load addresses in the image map.
Finished: 2 information, 0 warning and 1 error messages.
".\Objects\wujique.axf" - 1 Error(s), 0 Warning(s).
```

错误，
__aeabi_assert是C语言库中的函数，我们工程选用了微库，不包含assert功能。



可以去掉微库的钩，选择使用全库。

如果由于其他限制，无法使用全库，怎么办？
我们现在的工程已经用微库调试很久了，如果现在改用全库，很可能会出现各种问题。
而且，assert并不是必须的。

修改lfs定义

在lfs_util.h文件中有宏定义如下：

```
// Runtime assertions
#ifndef LFS_NO_ASSERT
#define LFS_ASSERT(test) assert(test)
#else
#define LFS_ASSERT(test)
#endif
```

把宏定义改为空操作，如下：

```
// Runtime assertions
#ifndef LFS_NO_ASSERT
//#define LFS_ASSERT(test) assert(test)
#define LFS_ASSERT(test)
#else
#define LFS_ASSERT(test)
#endif
```

重新编译就可以通过了。

lfs_util.h这个头文件，其实就是配置lfs相关函数的宏定义文件。
主要包含3个功能的定义：

> 断言
> LOG（错误和警告信息）
> 内存分配

我们的系统没有使用C库的内存分配，因此，LFS的内存分配也要修改，不使用库的。

```
/*
        lfs内存分配，默认使用c库的分配。
        我们的工程有自己的内存分配
*/
#include "alloc.h"

// Allocate memory, only used if buffers are not provided to littlefs
static inline void *lfs_malloc(size_t size) {
#ifndef LFS_NO_MALLOC
    //return malloc(size);
    return wjq_malloc(size);
#else
    (void)size;
    return NULL;
#endif
}

// Deallocate memory, only used if buffers are not provided to littlefs
static inline void lfs_free(void *p) {
#ifndef LFS_NO_MALLOC
    //free(p);
         wjq_free(p);
#else
    (void)p;
#endif
}
```

将调试信息用的printf改为wjq_log。

```
#include "wujique_log.h"

// Logging functions
#ifndef LFS_NO_DEBUG
#define LFS_DEBUG(fmt, ...) \
    wjq_log(LOG_INFO, "lfs debug:%d: " fmt "\n", __LINE__, __VA_ARGS__)
#else
#define LFS_DEBUG(fmt, ...)
#endif

#ifndef LFS_NO_WARN
#define LFS_WARN(fmt, ...) \
    wjq_log(LOG_ERR, "lfs warn:%d: " fmt "\n", __LINE__, __VA_ARGS__)
#else
#define LFS_WARN(fmt, ...)
#endif

#ifndef LFS_NO_ERROR
#define LFS_ERROR(fmt, ...) \
    wjq_log(LOG_ERR, "lfs error:%d: " fmt "\n", __LINE__, __VA_ARGS__)
#else
#define LFS_ERROR(fmt, ...)
#endif
```

重新编译，通过，没错误。

移植测试案例

将README.md中的例子拷贝到sys_littlefs.c，浏览，例子大概有以下问题：

- 变量冲突
  因为我们的工程已经包含了两个文件系统：fatfs、spifs。
  下面两个变量的命名方式很容易发生冲突，

```
lfs_t lfs;
lfs_file_t file;
```

- 接口不符合

我们的spi flash驱动是面向对象的，支持多个FLASH的操作。
但是在哪个flash上创建lfs文件系统，是唯一的。
请看lfs的配置，函数指针是操作固定FLASH的函数。

```
// configuration of the filesystem is provided by this struct
const struct lfs_config cfg = {
    // block device operations
    .read  = user_provided_block_device_read,
    .prog  = user_provided_block_device_prog,
    .erase = user_provided_block_device_erase,
    .sync  = user_provided_block_device_sync,

    // block device configuration
    .read_size = 16,
    .prog_size = 16,
    .block_size = 4096,
    .block_count = 128,
    .lookahead = 128,
};
```

因此，需要将我们的函数封装为固定FLASH的函数。
如果需要在多个FLASH上建立lfs，就需要封装多套函数。
当然，你也可以修改lfs支持多flash，目前还不熟悉lfs，先不要做大修改为好。

通过查看函数这4个函数指针的定义，可见，这些函数的参数，给我们的定义也不完全一致，同样需要修改。
最后我们重定义后的接口如下：

```c
#define LFS_FLASH_NAME "board_spiflash"
/*
        flash 接口,
        所有函数默认返回0，也就是说，默认操作FLASH成功。
*/

// Read a region in a block. Negative error codes are propogated
// to the user.
int user_provided_block_device_read(const struct lfs_config *c, lfs_block_t block,
                lfs_off_t off, void *buffer, lfs_size_t size)
{
        DevSpiFlashNode *node;

        node = dev_spiflash_open(LFS_FLASH_NAME);
        if(node == NULL)
        {
                        return -1;
        }

        dev_spiflash_read(node, block* c->block_size + off, size, buffer);

        dev_spiflash_close(node);

        return 0;
}

// Program a region in a block. The block must have previously
// been erased. Negative error codes are propogated to the user.
// May return LFS_ERR_CORRUPT if the block should be considered bad.
int user_provided_block_device_prog(const struct lfs_config *c, lfs_block_t block,
                lfs_off_t off, const void *buffer, lfs_size_t size)
{
        DevSpiFlashNode *node;

        node = dev_spiflash_open(LFS_FLASH_NAME);
        if(node == NULL)
        {
                        return -1;
        }

        dev_spiflash_write(node, block* c->block_size + off, size, buffer);

        dev_spiflash_close(node);

        return 0;
}

// Erase a block. A block must be erased before being programmed.
// The state of an erased block is undefined. Negative error codes
// are propogated to the user.
// May return LFS_ERR_CORRUPT if the block should be considered bad.
int user_provided_block_device_erase(const struct lfs_config *c, lfs_block_t block)
{
        DevSpiFlashNode *node;
```

```
        node = dev_spiflash_open(LFS_FLASH_NAME);
        if(node == NULL)
        {
                        return -1;
        }

        dev_spiflash_erase(node, block* c->block_size);

        dev_spiflash_close(node);

        return 0;
}

// Sync the state of the underlying block device. Negative error codes
// are propogated to the user.
int user_provided_block_device_sync(const struct lfs_config *c)
{
        return 0;
}
```

- 修改FLASH配置

```
// block device configuration
.read_size = 256,
.prog_size = 256,
.block_size = 4096,
.block_count = 128,
.lookahead = 256,
```

将测试程序添加到main函数中，先挂载文件系统，再测试。

```
sys_lfs_mount();
lfs_test();
```

重新编译后下载到设备中，启动后，出现调试信息，
有错误，是因为在flash上没有创建LFS文件系统。

```
lfs error:574: Corrupted dir pair at 0 1
lfs error:2353: Invalid superblock at 0 1
boot_count: 1
```

复位设备，看到调试信息：

```
boot_count: 2
```

对设备断电，调试信息只看到boot_count计数，并没有出现错误，说明在flash上创建的文件系统没有问题。

最终移植测试代码请查看附件

LFS主要算法

待补充

END