GigaDevice Semiconductor Inc.

Arm® Cortex®-M3/4/23/33 32-bit MCU

应用笔记

AN015

目录

图索引

表索引

1. 简介

LittleFS 是 Arm®面向嵌入式设备推出的一款开源的小型文件系统，具有抗掉电，动态磨损均衡，占用 RAM/ROM 少等特点，适合用于 IOT 嵌入式设备中管理 SPI-flash。具体介绍可见 ***https://github.com/ARMmbed/littlefs***。

本文介绍了 LittleFS 移植到 GD32 工程下的方法，并对文件系统的读写功能进行了测试。

2. LittleFS 移植

2.1. LittleFS 移植平台

本文介绍的 LittleFS 移植平台为 GD32F450Z-EVAL 开发板，GD32F450Z-EVAL 开发板上外挂了一颗 SPI-flash，该 SPI-flash 的型号为 GD25Q16。LittleFS 移植的 IDE 平台为 KEIL4，在 GD32F450Z 的 SPI_QSPI_Flash 工程上进行代码移植，工程版本为 V2.0.0。

LittleFS 的文件比较简单，只需要 lfs.c、lfs.h、lfs_util.c 和 lfs_util.h 四个文件。本文移植的 LittleFS 版本为 LFS_VERSION 0x00020002。LittleFS 的版本信息可从 lfs.h 文件中获取，其版本信息的如*图 0-1. LittleFS 的版本信息*。

图 0-1. LittleFS 的版本信息

```
/// Version info ///

// Software library version
// Major (top-nibble), incremented on backwards incompatible changes
// Minor (bottom-nibble), incremented on feature additions
#define LFS_VERSION 0x00020002
#define LFS_VERSION_MAJOR (0xffff & (LFS_VERSION >> 16))
#define LFS_VERSION_MINOR (0xffff & (LFS_VERSION >>  0))

// Version of On-disk data structures
// Major (top-nibble), incremented on backwards incompatible changes
// Minor (bottom-nibble), incremented on feature additions
#define LFS_DISK_VERSION 0x00020000
#define LFS_DISK_VERSION_MAJOR (0xffff & (LFS_DISK_VERSION >> 16))
#define LFS_DISK_VERSION_MINOR (0xffff & (LFS_DISK_VERSION >>  0))
```

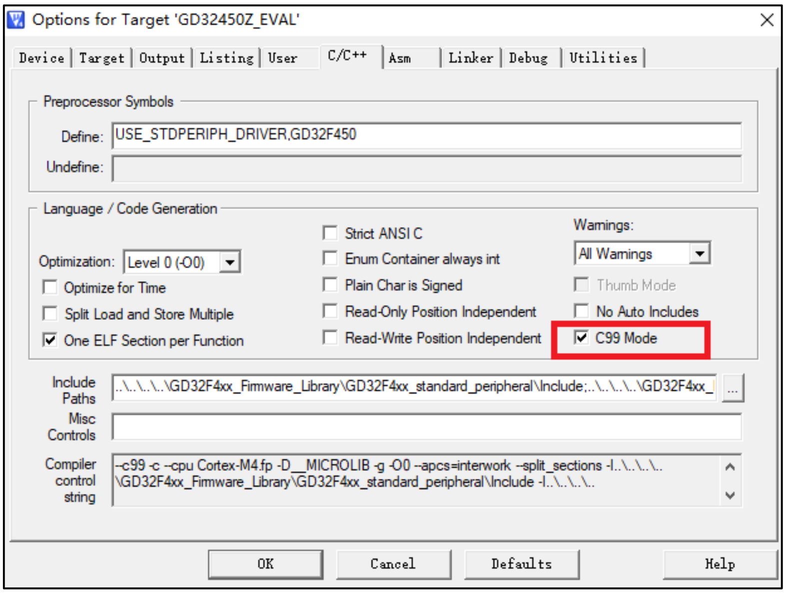
2.2. 添加 LittleFS 源码文件

本文介绍的移植方式基于 GD32F450Z 的 SPI_QSPI_Flash 工程，首先将 LittleFS 文件拷贝至 GD32F4xx_Demo_Suites_V2.1.0\GD32450Z_EVAL_Demo_Suites\Projects\SPI_QSPI_Flash\Soft_Drive 文件下中。然后打开工程，在工程中添加 lfs.c 和 lfs_util.c 两个文件。

2.3. IDE 配置

LittleFS 在使用 KEIL4 编译器时，必须配置支持 C99 标准。其选项配置如*图 0-2. KEIL4 配置 C99 标准*。

图 0-2. KEIL4 配置 C99 标准



在配置 C99 标准后，编译工程，测试 LittleFS 源码是否能编译通过。编译结果如*图 0-3. 初次编译 LittleFS 源码*，显示__aeabi_assert 函数未定义。因为 GD32 的工程都选择使用了微库，不包含 assert 功能，当 KEIL4 打开优化等级进行编译时，编译报错。

图 0-3. 初次编译 LittleFS 源码

```
Build target 'GD32450Z_EVAL'
linking...
.\output\GD32450Z_EVAL.axf: Error: L6218E: Undefined symbol __aeabi_assert (referred from lfs.o).
Not enough information to list image symbols.
Finished: 1 information, 0 warning and 1 error messages.
".\output\GD32450Z_EVAL.axf" - 1 Error(s), 0 Warning(s).
Target not created
```

在 lfs_util.h 文件中有关于 assert 功能的宏定义，如*图 0-4. LittleFS 中 assert 功能的宏定义所示*。

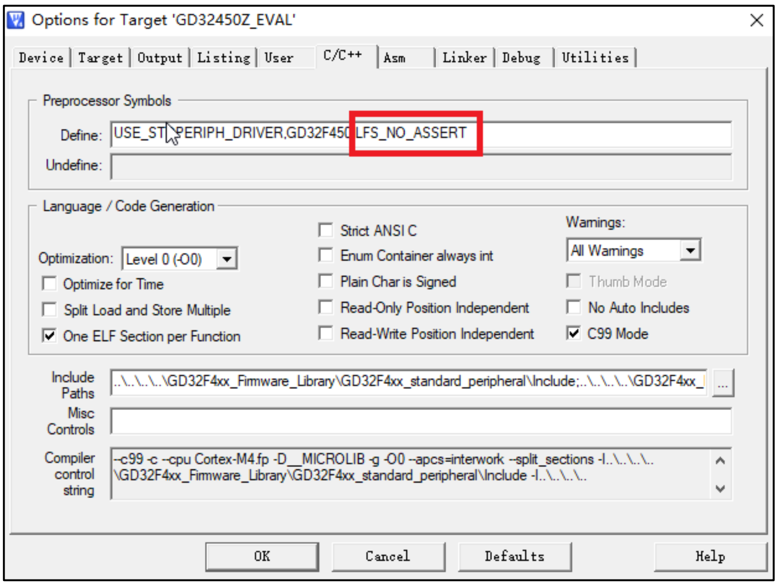
图 0-4. LittleFS 中 assert 功能的宏定义

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

由于 assert 功能不是必须的，因此可通过两种方式解决上述问题。

1. 在 KEIL4/KEIL5 中增加宏定义。添加方式如*图 0-5. KEIL4 中添加 LFS_NO_ASSERT 宏定义*所示。

图 0-5. KEIL4 中添加 LFS_NO_ASSERT 宏定义



2. 修改 lfs_util.h 文件中的宏定义，把宏定义改为空操作，修改结果如***图 0-6. lfs_util.h 文件中修改 assert 宏定义***。

图 0-6. lfs_util.h 文件中修改 assert 宏定义

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

完成上述相关操作后，再点击编译，则编译成功通

2.4. LittleFS 参数配置

在 lfs.h 中定义了 LittleFS 的配置参数结构体 struct lfs_config，LittleFS 在管理 SPI-flash 时，需要根据实际的 SPI-flash 型号进行参数配置。本文的移植例程使用 SPI-flash 的型号为 GD25Q16。其相关的参数配置如***表 0-1. LittleFS 的配置参数***所示。

表 0-1. LittleFS 的配置参数

```
/*!
    \brief      config the block device interface
    \param[in]  none
    \param[out] none
    \retval     none
*/
void lfs_config(void)
{
```

```
    /* block device operations */
    g_lfs_cfg.read = block_device_read;        //link the block_device_read function
    g_lfs_cfg.prog = block_device_prog;        //link the block_device_prog function
    g_lfs_cfg.erase = block_device_erase;      //link the block_device_sync function
    g_lfs_cfg.sync = block_device_sync;        //link the block_device_sync function

    /* block device configuration */
    g_lfs_cfg.read_size = 256;                 //config read data size for each block(256 byte)
    g_lfs_cfg.prog_size = 256;                 //config write data size for each block(256 byte)
    g_lfs_cfg.block_size = 4096;               //config the block size(4096 byte

    g_lfs_cfg.cache_size = 256;                //Must be a multiple of the read and program sizes
    g_lfs_cfg.block_count = 1024;              //the total of block
    g_lfs_cfg.lookahead_size = 128;            //Predictive depth for block allocation:1024/8=128
    g_lfs_cfg.block_cycles = 500;              //Set to -1 to disable block-level wear-leveling
}
```

在结构体 lfs_config 中，定义了 4 个函数指针：int (*read)、int (*prog)、int (*erase)和 int (*sync)，被调用的接口函数需要用户自己定义。本文基于 GD25Q16 SPI-flash 的接口函数如*表 0-2. LittleFS 基于 GD25Q16 SPI-flash 的接口函数定义*所示。

表 0-2. LittleFS 基于 GD25Q16 SPI-flash 的接口函数定义

```
/*!
    \brief        read the data from spi flash block
    \param[in]    *c : the lfs_config struct pointer
    \param[in]    block: the number of block
    \param[in]    off: the offset in block
    \param[in]    buffer: the read data buffer
    \param[in]    size: the size of read data
    \param[out] none
    \retval       none
*/
int32_t block_device_read(const struct lfs_config *c, lfs_block_t block,
    lfs_off_t off, void *buffer, lfs_size_t size)
{
    /* read the data from spi flash */
    spi_flash_buffer_read((uint8_t*) buffer,(block * (c->block_size) + off),size);
    return 0;
}


/*!
    \brief        write the data from spi flash block
    \param[in]    *c : the lfs_config struct pointer
    \param[in]    block: the number of block
```

```
    \param[in]   off: the offset in block
    \param[in]   buffer: the write data buffer
    \param[in]   size: the size of write data
    \param[out] none
    \retval       none
*/
int32_t block_device_prog(const struct lfs_config *c, lfs_block_t block,
    lfs_off_t off, const void *buffer, lfs_size_t size)
{
    /* write the data to spi flash */
    spi_flash_buffer_write((uint8_t*)buffer, ((block) * (c->block_size) + off), size);
    return 0;
}

/*!
    \brief       erase the spi flash block
    \param[in]   *c : the lfs_config struct pointer
    \param[in]   block: the number of block
    \param[out] none
    \retval       none
*/
int32_t block_device_erase(const struct lfs_config *c, lfs_block_t block)
{
    /* erase the sector of spi flash */
    spi_flash_sector_erase(block * (c->block_size));

    return 0;
}

/*!
    \brief       Sync the state of the underlying block device.
    \param[in]   none
    \param[out] none
    \retval       none
*/
int32_t block_device_sync(const struct lfs_config *c)
{
    /* no operation */
    return 0;
}
```

3. LittleFS 功能测试

本章节介绍将对移植 LittleFS 后进行读写功能的测试，并给出测试 demo。在测试 LittleFS 功能前需要先挂载文件系统，挂载 LittleFS 的代码如*表 0-3. LittleFS 挂载代码*。

表 0-3. LittleFS 挂载代码

```
/*!
    \brief      mount the filesystem
    \param[in]  none
    \param[out] none
    \retval     none
*/
void sys_lfs_mount(void)
{
    lfs_config();
    /* mount the filesystem */
    int err = lfs_mount(&g_lfs, &g_lfs_cfg);
    /* reformat if we can't mount the filesystem£¬this should only happen on the first boot */
    if (err) {
        /* format a block device with the littlefs */
        lfs_format(&g_lfs, &g_lfs_cfg);
        /* mount the filesystem */
        lfs_mount(&g_lfs, &g_lfs_cfg);
    }
}
```

3.1. LittleFS 掉电保护功能测试

掉电保护功能是 LittleFS 的一个优点，如*表 0-4. LittleFS 掉电保护功能测试代码*是掉电保护功能测试的代码。

表 0-4. LittleFS 掉电保护功能测试代码

```
/*!
    \brief      lilltleFS power-off protection test
    \param[in]  none
    \param[out] none
    \retval     none
*/
void lfs_power_off_protection_test(void)
{
    uint32_t boot_count = 0;

    /* open the file */
```

```
    lfs_file_open(&g_lfs, &g_lfs_file, "boot_count", LFS_O_RDWR | LFS_O_CREAT);
    /* read the data */
    lfs_file_read(&g_lfs, &g_lfs_file, &boot_count, sizeof(boot_count));

    /* update boot count */
    boot_count += 1;
    /* write to the beginning of the file */
    lfs_file_rewind(&g_lfs, &g_lfs_file);
    lfs_file_write(&g_lfs, &g_lfs_file, &boot_count, sizeof(boot_count));

    /* remember the storage is not updated until the file is closed successfully */
    lfs_file_close(&g_lfs, &g_lfs_file);

    /* release any resources */
    lfs_unmount(&g_lfs);

    /* print the boot count */
    printf("boot_count:%d\n", boot_count);

}
```

每次 main 运行时都会更新名为“boot”count 的文件。程序可以随时中断，而不会丢失启动次数的记录，也不会损坏文件系统。进行多少掉电测试，其测试结果如*图 0-7. LittleFS 掉电保护功能测试*所示。

图 0-7. LittleFS 掉电保护功能测试

上图所示，在第一次启动挂载文件系统时，会发生不能挂装载文件系统的情况。此时需要重新格式化，再挂载。

3.2. LittleFS 更新文件数据测试

更新文件数据的测试，主要是对同一文件的多次写入数据，并通过串口打印文件内容。再利用 LittleFS 的裁剪文件数据的功能，删除不需要的数据。其测试例程如***表 0-5. LittleFS 更新文件数据代码***所示。

表 0-5. LittleFS 更新文件数据代码

```
/*!
    \brief       update the file in random positin
    \param[in]   write_buffer1: the data1 for write to file
    \param[in]   write_buffer2: the data2 for write to file
    \param[in]   read_buffer1: the data1 for read from file
    \param[in]   read_buffer2: the data2 for read from file
    \param[in]   byte_cnt1: the count of write data1
    \param[in]   byte_cnt2: the count of write data2
    \param[out]  none
    \retval      none
*/
void lfs_update_file(const uint8_t* write_buffer1,const uint8_t* write_buffer2,
```

```
                    uint8_t* read_buffer1, uint8_t* read_buffer2,
                    uint32_t byte_cnt1, uint32_t byte_cnt2)
{
    /* open the "E:\\my_test_file" file */
    lfs_file_open(&g_lfs, &g_lfs_file, "E:\\my_test_file", LFS_O_RDWR | LFS_O_CREAT);
    /* write the data to the "E:\\my_test_file" file */
    lfs_file_write(&g_lfs, &g_lfs_file, write_buffer1, byte_cnt1);
    /* the starting position in the file for writing or reading */
    lfs_file_seek(&g_lfs, &g_lfs_file,0,LFS_SEEK_SET);
    /* read the data from the "E:\\my_test_file" file */
    lfs_file_read(&g_lfs, &g_lfs_file, read_buffer1, byte_cnt1);

    printf("E:\\my_test_file\n");
    printf("%s\n", read_buffer1);
    /* the starting position in the file for writing or reading */
    lfs_file_seek(&g_lfs, &g_lfs_file,byte_cnt1,LFS_SEEK_SET);
    /* write the data to the "E:\\my_test_file" file */
    lfs_file_write(&g_lfs, &g_lfs_file, write_buffer2, byte_cnt2);
    /* read from the beginning of the file */
    lfs_file_rewind(&g_lfs, &g_lfs_file);
    /* read the data from the "E:\\my_test_file" file */
    lfs_file_read(&g_lfs, &g_lfs_file, read_buffer1, byte_cnt1+byte_cnt2);

    printf("E:\\my_test_file\n");
    printf("%s\n", read_buffer1);

    lfs_file_seek(&g_lfs, &g_lfs_file,0,LFS_SEEK_SET);
    /* Intercept part of the data of the file and retain y bytes of data */
    lfs_file_truncate(&g_lfs, &g_lfs_file,byte_cnt2);

    lfs_file_read(&g_lfs, &g_lfs_file, read_buffer2, byte_cnt2);
    /* remember the storage is not updated until the file is closed successfully */
    lfs_file_close(&g_lfs, &g_lfs_file);
    /*release any resources we were using*/
    lfs_unmount(&g_lfs);

    printf("E:\\my_test_file\n");
    printf("%s\n", read_buffer2);

    /* delete the file */
    lfs_remove(&g_lfs,"E:\\my_test_file");
}
```

测试结果如*图 0-8. 更新文件数据测试*所示。由图中所示，创建了"E:\\my_test_file"文件，对该

文件的内容进行了更新，最后一次的打印结果为裁剪部分数据后，文件保留的数据。

图 0-8. 更新文件数据测试

4. 版本历史

表 0-6. 版本历史

版本号.	说明	日期
1.0	首次发布	2021 年 12 月 13 日

Important Notice