GigaDevice Semiconductor Inc.

Arm® Cortex®-M3/4/23/33 32-bit MCU

Application Note
AN015

Table of Contents

Table of Contents 2
List of Figures 3
List of Table 4
1. Introduction 5
2. LittleFS transplantation 6
2.1. LittleFS transplanted platform 6
2.2. Add the LittleFS source file 6
2.3. IDE configuration 6
2.4. LittleFS parameter configuration 8
3. LittleFS functional test 11
3.1. LittleFS power failure protection function test 11
3.2. LittleFS update file data test 13
4. Revision history 16

List of Figures

Figure 2-1. LittleFS version information 6
Figure 2-2. KEIL4 configures C99 standard 7
Figure 2-3. Compile LittleFS source code for the first time 7
Figure 2-4. Macro definition of assert function in LittleFS 7
Figure 2-5. Add LFS_NO_ASSERT macro definition in KEIL4 8
Figure 2-6. Modify the assert macro definition in the lfs_util.h file 8
Figure 3-1. LittleFS power failure protection function test 13
Figure 3-2. Update file data test 15

List of Table

Table 2-1. LittleFS configuration parameters 8
Table 2-2. LittleFS is based on the interface function definition of GD25Q16 SPI-flash 9
Table 3-1. LittleFS mount code 11
Table 3-2. LittleFS power failure protection function test code 11
Table 3-3. LittleFS update file data code 13
Table 4-1. Revision history 16

1. Introduction

LittleFS is an open source small file system launched by Arm® for embedded devices. It has the characteristics of anti-power failure, dynamic wear leveling, and less RAM/ROM occupation. It is suitable for managing SPI-flash in IOT embedded devices. The specific introduction can be found at ***https://github.com/ARMmbed/littlefs.***

This article introduces the method of porting LittleFS to GD32 project, and tests the read and write functions of the file system.

2. LittleFS transplantation

2.1. LittleFS transplanted platform

The LittleFS transplantation platform introduced in this article is the GD32F450Z-EVAL board. An SPI-flash is attached to the GD32F450Z-EVAL board. The SPI-flash model is GD25Q16. Porting of LittleFS uses KEIL4, and the code is ported on the SPI_QSPI_Flash project of GD32F450Z, and the project version is V2.0.0.

The file of LittleFS is very simple, only four files of “lfs.c”, “lfs.h”, “lfs_util.c” and “lfs_util.h”. The version of LittleFS transplanted in this article is LFS_VERSION 0x00020002. The version information of LittleFS can be obtained from the “lfs.h” file, and the version information is shown in *Figure 2-1. LittleFS version information*.

Figure 2-1. LittleFS version information

```
19  /// Version info ///
20
21  // Software library version
22  // Major (top-nibble), incremented on backwards incompatible changes
23  // Minor (bottom-nibble), incremented on feature additions
24  #define LFS_VERSION 0x00020002
25  #define LFS_VERSION_MAJOR (0xffff & (LFS_VERSION >> 16))
26  #define LFS_VERSION_MINOR (0xffff & (LFS_VERSION >>  0))
27
28  // Version of On-disk data structures
29  // Major (top-nibble), incremented on backwards incompatible changes
30  // Minor (bottom-nibble), incremented on feature additions
31  #define LFS_DISK_VERSION 0x00020000
32  #define LFS_DISK_VERSION_MAJOR (0xffff & (LFS_DISK_VERSION >> 16))
33  #define LFS_DISK_VERSION_MINOR (0xffff & (LFS_DISK_VERSION >>  0))
```

2.2. Add the LittleFS source file

The transplantation method introduced in this article is based on the SPI_QSPI_Flash project of GD32F450Z. First, copy the LittleFS file to the follow folder:

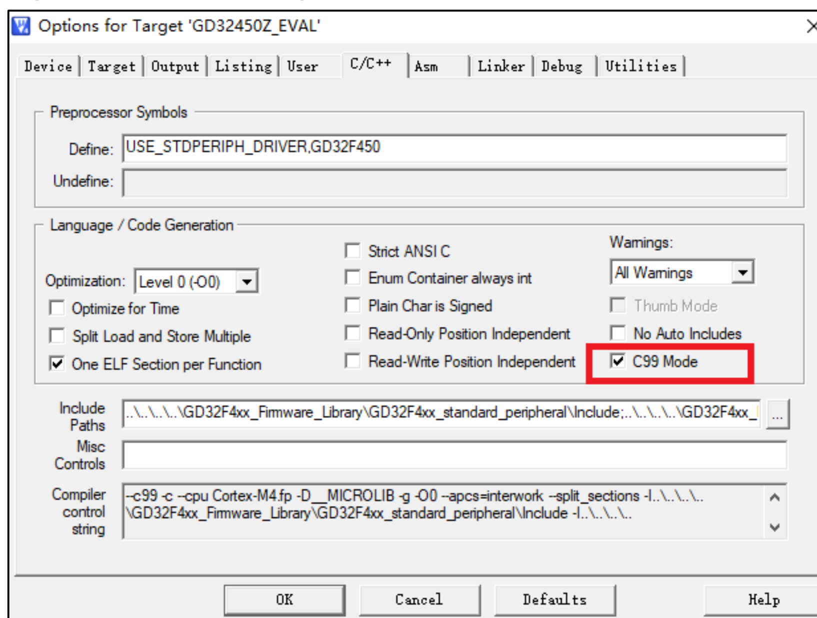
GD32F4xx_Demo_Suites_V2.1.0\GD32450Z_EVAL_Demo_Suites\Projects\SPI_QSPI_Flash\Soft_Drive.

Then open the project and add two files lfs.c and lfs_util.c to the project.

2.3. IDE configuration

LittleFS must be configured to support the C99 standard when using KEIL4 to compiler. The option configuration is shown in *Figure 2-2. KEIL4 configures C99 standard*.

Figure 2-2. KEIL4 configures C99 standard



After configuring the C99 standard, compile the project and test whether the LittleFS source code can be compiled successfully. The compilation result is shown in ***Figure 2-3. Compile LittleFS source code for the first time***, showing that the __aeabi_assert function is undefined. Because GD32 projects all choose to use micro-library and do not include the assert function, when KEIL4 opens the optimization level for compilation, compilation errors are reported.

Figure 2-3. Compile LittleFS source code for the first time

```
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
```

There is a macro definition about the assert function in the "lfs_util.h" file, as shown in ***Figure 2-4. Macro definition of assert function in LittleFS***.

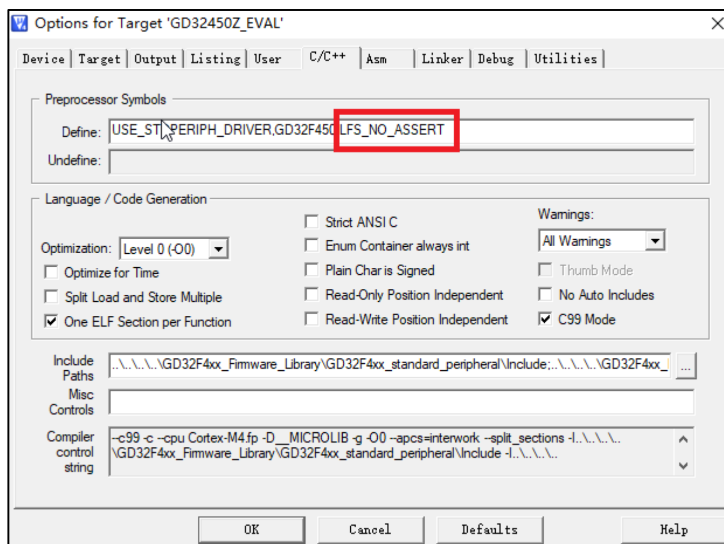
Figure 2-4. Macro definition of assert function in LittleFS

```
84 // Runtime assertions
85 #ifndef LFS_NO_ASSERT
86 #define LFS_ASSERT(test) assert(test)
87 #else
88 #define LFS_ASSERT(test)
89 #endif
```

Since the assert function is not necessary, the above problems can be solved in two ways.

1. Add macro definition in KEIL4/KEIL5. The adding method is shown in ***Figure 2-5. Add LFS_NO_ASSERT macro definition in KEIL4***.

Figure 2-5. Add LFS_NO_ASSERT macro definition in KEIL4



Modify the macro definition in the lfs_util.h file and change the macro definition to a no-op. The modification results are as ***Figure 2-6. Modify the assert macro definition in the lfs_util.h file***.

Figure 2-6. Modify the assert macro definition in the lfs_util.h file

```
84  // Runtime assertions
85  #ifndef LFS_NO_ASSERT
86  //#define LFS_ASSERT(test) assert(test)
87  #define LFS_ASSERT(test)
88  #else
89  #define LFS_ASSERT(test)
90  #endif
```

After completing the above related operations, and then click compile, the compile successfully

2.4. LittleFS parameter configuration

The configuration parameter structure “struct lfs_config” of LittleFS is defined in “lfs.h”. When LittleFS manages SPI-flash, it is necessary to configure the parameters according to the actual SPI-flash. The transplantation example of this article uses GD25Q16 SPI-flash. The related parameter configuration is shown in ***Table 2-1. LittleFS configuration parameters***.

Table 2-1. LittleFS configuration parameters

```
/*!
    \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
}
```

In the structure lfs_config, four function pointers are defined: int (*read), int (*prog), int (*erase) and int (*sync) and the interface function to be called needs to be completed by the user. This article is based on the interface functions of GD25Q16 SPI-flash as shown in ***Table 2-2. LittleFS is based on the interface function definition of GD25Q16 SPI-flash***.

Table 2-2. LittleFS is based on the interface function definition of GD25Q16 SPI-flash

```
/*!
    \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 functional test

This chapter introduces the test of read and write function after porting LittleFS, and gives a test demo.Before testing the LittleFS function, it is necessary to mount the file system. The code of mounting LittleFS is as ***Table 3-1. LittleFS mount code.***

Table 3-1. LittleFS mount code

```
/*!
    \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 power failure protection function test

The power-down protection function is an advantage of LittleFS. The code for the power-down protection function test refers to ***Table 3-2. LittleFS power failure protection function test code***.

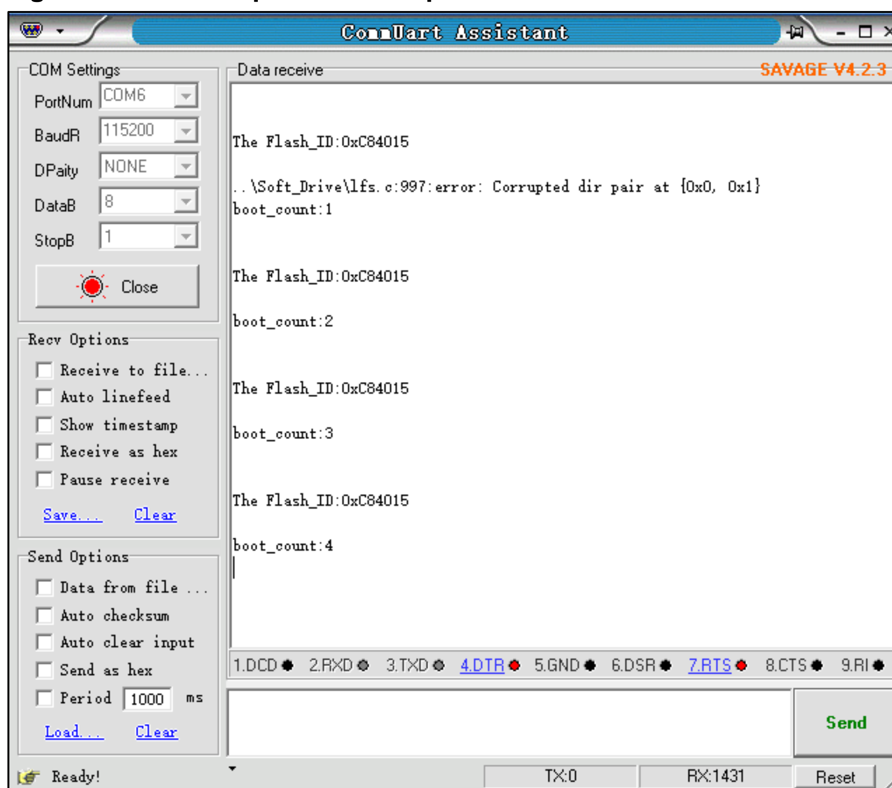
Table 3-2. LittleFS power failure protection function test code

```
/*!
    \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);

}
```

The file named "boot_count" is updated every time the main function runs. The program can be interrupted at any time, without losing the record of the number of starts, and without damaging the file system. How many power-down tests are performed, and the test results are shown in ***Figure 3-1. LittleFS power failure protection function test***.

Figure 3-1. LittleFS power failure protection function test



As shown in the figure above, when the file system is mounted for the first time, the file system can not be mounted. At this time, it needs to be reformatted and then mounted.

3.2. LittleFS update file data test

The test of updating file data is mainly to write data to the same file multiple times and print the file content through the serial port. Then use LittleFS's function of cropping file data to delete unnecessary data. The test demo is as ***Table 3-3. LittleFS update file data code***.

Table 3-3. LittleFS update file data code

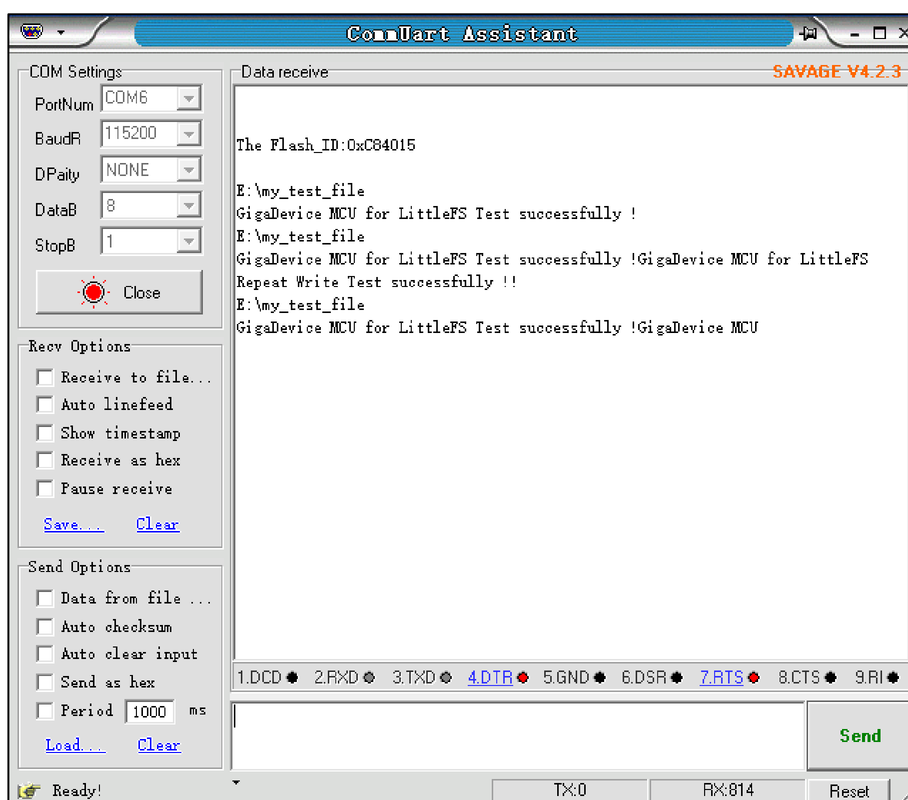
```
/*!
    \brief      read the data from spi flash block
    \param[in]  *c : the lfs_config struct pointer
    \param[in]  block: the number of block
    \param[in]  off: the offset in block
    \param[in]  buffer: the read data buffer
    \param[in]  size: the size of read data
    \param[out] none
    \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;
}
```

The test result is shown in ***Figure 3-2. Update file data test***. As shown in the figure, the "E:\\my_test_file" file is created, and the content of the file is updated. The last print result is the data retained by the file after cutting part of the data.

Figure 3-2. Update file data test



4. Revision history

Table 4-1. Revision history

Revision No.	Description	Date
1.0	Initial Release	Dec.13 2021

Important Notice

This document is the property of GigaDevice Semiconductor Inc. and its subsidiaries (the "Company"). This document, including any product of the Company described in this document (the “Product”), is owned by the Company under the intellectual property laws and treaties of the People’s Republic of China and other jurisdictions worldwide. The Company reserves all rights under such laws and treaties and does not grant any license under its patents, copyrights, trademarks, or other intellectual property rights. The names and brands of third party referred thereto (if any) are the property of their respective owner and referred to for identification purposes only.

The Company makes no warranty of any kind, express or implied, with regard to this document or any Product, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Company does not assume any liability arising out of the application or use of any Product described in this document. Any information provided in this document is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Except for customized products which has been expressly identified in the applicable agreement, the Products are designed, developed, and/or manufactured for ordinary business, industrial, personal, and/or household applications only. The Products are not designed, intended, or authorized for use as components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, atomic energy control instruments, combustion control instruments, airplane or spaceship instruments, transportation instruments, traffic signal instruments, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or Product could cause personal injury, death, property or environmental damage ("Unintended Uses"). Customers shall take any and all actions to ensure using and selling the Products in accordance with the applicable laws and regulations. The Company is not liable, in whole or in part, and customers shall and hereby do release the Company as well as it’s suppliers and/or distributors from any claim, damage, or other liability arising from or related to all Unintended Uses of the Products. Customers shall indemnify and hold the Company as well as it’s suppliers and/or distributors harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of the Products.

Information in this document is provided solely in connection with the Products. The Company reserves the right to make changes, corrections, modifications or improvements to this document and Products and services described herein at any time, without notice.

© 2021 GigaDevice – All rights reserved