UM1721
User manual

Developing Applications on STM32Cube with FatFs

Introduction

The STMCube™ initiative was originated by STMicroelectronics to ease developers life by reducing development efforts, time and cost. STM32Cube covers the STM32 portfolio.

STM32Cube Version 1.x includes:

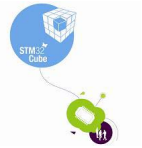
- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (namely, STM32CubeF4 for STM32F4 series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, TCP/IP, Graphics
  - All embedded software utilities coming with a full set of examples.

A File System is the way in which files ares named and where they are placed logically for storage and retrieval. Its primary objective is to manage access to the data of files, and to manage the available space of the device(s) which contain it. Using a file system allows user to ensure reliability and to organize data in an efficient manner.

This user manual is intended for developers who use STM32Cube firmware on STM32 microcontrollers. It provides a full description of how to use the STM32Cube firmware components with a generic FAT File System (FatFs); this user manual comes also with description of a set of examples based on common FatFs provided APIs.

Please refer to the release notes of the STM32Cube firmware package to know the version of FatFs firmware component used.

This document is applicable to all STM32 devices; however for simplicity reason, the STM32F4xx devices and STM32CubeF4 are used as reference platform. To know more about supported physical media disk and the examples implementation on your STM32 device, please refer to the readme file provided within the associated STM32Cube FW package.



Contents

List of tables

List of figures

Acronyms and definitions

Table 1. Acronyms and definitions

Acronym	Definition
ANSI	American National Standards Institute
API	Application Programming Interface
BPB	BIOS Parameter Block
BSP	Board Support Package
CPU	Central Processing Unit
CMSIS	Cortex™ Microcontroller Software Interface Standard
DBCS	Double Byte Char String
DOS	Disk Operating System
EFI	Extensible Firmware Interface
FAT	File Allocation Table
HAL	Hardware Abstraction Table
LFN	Long File Name
MBR	Master Boot Record
MSD	Micro Secure Digital
OEM	Original Equipment Manufacturer
RAM	Random Access Memory
RTC	Real Time Clock
RTOS	Real Time Operating System
SD	Secure Digital
SDRAM	Synchronous Dynamic Random Access Memory
SFN	Short File Name
SRAM	static Random Access Memory
USB	Universal serial Bus

1 FAT File System overview

1.1 FAT overview

The File Allocation Table (FAT) file system was developed by Bill Gates and Marc McDonald. It is a format and some software which stores and organizes files on a storage device, such as a disk drive or a memory device. It is used to facilitate access to files and directories.

The FAT file system provides a way to time stamp when a file is created or changed and provides a way to identify the size of the file. This system provides a mechanism to store other attributes of a file, such as whether a file is read-only, whether the file should be hidden in a directory display, or whether a file should be archived at the next disk backup.

The FAT file system is ideal for removable Flash media used in consumer electronic devices, such as digital cameras, media players and Flash drives.

The FAT file system can be helpful in the following scenarios:

- Due to the backward compatibility of the FAT file system, users can employ memory stick media or floppy disks to transfer files between a consumer electronic device and a computer that uses an outdated operating system;
- The FAT file system lets users quickly remove files from electronic devices, as in professional broadcast media;
- The file system versions, FAT16 or FAT32, may be suitable for a hard disk drive volume.

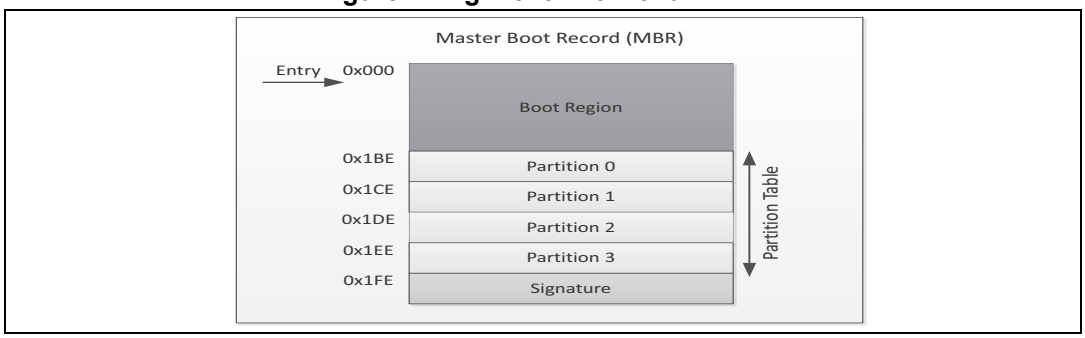
These versions would also be useful to a user who wants to boot a computer by using a floppy disk to access data (typically, system recovery tools) on a hard disk drive volume.

1.1.1 Master Boot Record

The Master Boot Record (MBR) is located on one or more sectors at the physical start of the device. The boot region of the MBR contains DOS boot loader code, which is written when the device is formatted (but is not otherwise used by the Dynamic C FAT file system). The partition table follows the boot region. It contains four 16-byte entries, which allow up to four partitions on the device.

Partition table entries contain some critical information: the partition type (Dynamic C FAT recognizes FAT12 and FAT16 partition types) and the partition's starting and ending sector numbers. There is also a field denoting the total number of sectors in the partition. If this number is zero, the corresponding partition is empty and available.

Figure 1. High level view of an MBR

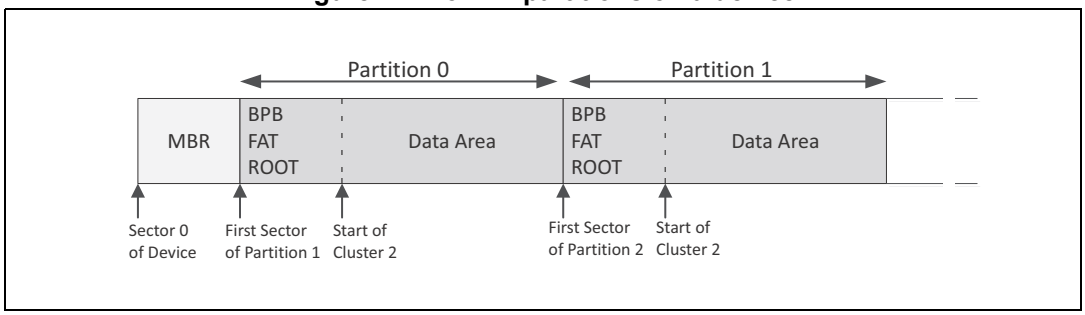


Some devices are formatted without an MBR and, therefore, have no partition table. This configuration is not currently supported in the Dynamic C FAT file system.

1.1.2 FAT partitions

The first sector of a valid FAT file system partition contains the BIOS Parameter Block (BPB), followed by the File Allocation Table (FAT), and then the Root Directory. The figure below shows a device with two FAT partitions.

Figure 2. Two FAT partitions on a device



BIOS Parameter Block

The fields of the BPB contain information describing the partition:

- The number of bytes per sector;
- The number of sectors per cluster;
- The total count of sectors on the partition;
- The number of root directory entries.

FAT Allocation Table

The file allocation table is the structure that gives the FAT file system its name. The FAT stores information about cluster assignments. A cluster is either assigned to a file, is available for use, or is marked as bad. A second copy of the FAT immediately follows the first one.

Root directory

The root directory has a predefined location and size. It has 512 entries of 32 bytes each. An entry in the root directory is either empty or contains a file or subdirectory name (in 8.3 format), file size, date and time of last revision and the starting cluster number for the file or subdirectory.

Data area

The data area takes up most of the partition. It contains file data and subdirectories. Note that the data area of a partition must, by convention, start at cluster 2.

For more details, refer to the Microsoft® EFI FAT32 File System Specification.

1.1.3 FAT license

The Microsoft Extensible Firmware Initiative FAT32 File System Specification, rev. 1.03, December 6, 2000, is available as an Office Word document (268 kBytes).

The download license agreement allows using the Microsoft EFI FAT32 File System Specification only in connection with a firmware implementation of the Extensible Firmware Initiative Specification, v. 1.0. If you plan to implement the FAT32 File System specification for other purposes, you must obtain an additional license from Microsoft.

For example, you must obtain an additional license in order to create a file system for reading, or reading and writing FAT32 in digital cameras recording to Flash media, in computer operating systems reading and writing internal/external hard disks or Flash media, or in set-top boxes reading FAT-formatted media.

For more details about FAT and applicable licenses and/or copyrights, refer to Microsoft web site.

2 FatFs File System

2.1 FatFs overview

FatFs is a generic FAT file system module for small embedded systems. The FatFs is written in compliance with ANSI C and completely separated from the disk I/O layer. Therefore it is independent of hardware architecture, and has the following features:

- Windows compatible FAT file system.
- Very small footprint for code and work area.
- Various configuration options:
  - Multiple volumes (physical drives and partitions).
  - Multiple ANSI/OEM code pages including DBCS.
  - Long file name support in ANSI/OEM or Unicode.
  - RTOS support.
  - Multiple sector size support.
  - Read-only, minimized API, I/O buffer and etc...
  - FAT sub-types: FAT12, FAT16 and FAT32.
  - Number of open files: Unlimited, depends on available memory.
  - Number of volumes: Up to 10.
  - File size: Depends on FAT specs. (up to 4G-1 bytes)
  - Volume size: Depends on FAT specs. (up to 2T bytes on 512 bytes/sector)
  - Cluster size: Depends on FAT specs. (up to 64K bytes on 512 bytes/sector)
  - Sector size: Depends on FAT specs. (up to 4K bytes)

2.2 FatFs architecture

FatFs module is a middleware which provides many functions to access the FAT volumes, such as f_open(), f_close(), f_read(), f_write(), etc (refer to ff.c).

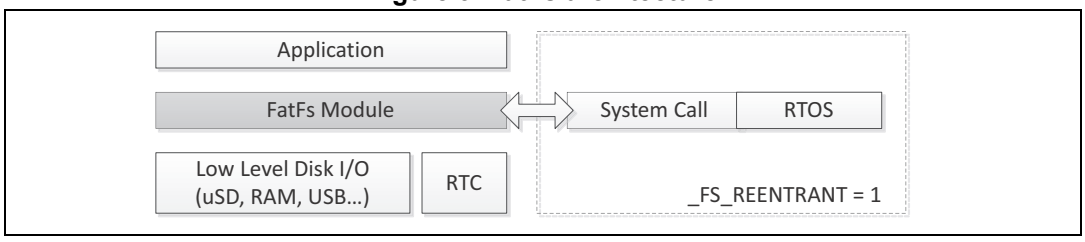
There is no platform dependence in this module, as long as the compiler is compliant with ANSI C.

A low level disk I/O module is used to read/write the physical drive,

An RTC module is used to get the current time.

The low level disk I/O and the RTC module are completely separate from the FatFs module. They must be provided by the user, which is the main task of porting FatFs module to other platforms.

Figure 3. FatFs architecture



2.3 FatFs license

Figure 4 is a copy of the FatFs license document included in the source codes.

Figure 4. FatFs license

```
/*----------------------------------------------------------------------------/
/  FatFs - FAT file system module  R0.10                     (C)ChaN, 2013
/-----------------------------------------------------------------------------/
/ FatFs module is a generic FAT file system module for small embedded systems.
/ This is a free software that opened for education, research and commercial
/ developments under license policy of following trems.
/
/  Copyright (C) 2013, ChaN, all right reserved.
/
/ * The FatFs module is a free software and there is NO WARRANTY.
/ * No restriction on use. You can use, modify and redistribute it for
/   personal, non-profit or commercial products UNDER YOUR RESPONSIBILITY.
/ * Redistributions of source code must retain the above copyright notice.
/
----------------------------------------------------------------------------/
```

Therefore FatFs license is one of the BSD-style licenses, but there is a big difference. Because FatFs is for embedded projects, the conditions for redistributions in binary form, such as embedded code, hex file and binary library, are not specified to increase its usability. The documentation of the distributions need not include about FatFs and its license document, and it may also. Of course FatFs is compatible with the projects under GNU GPL. When redistribute it with any modification, the license can also be changed to GNU GPL or BSD-style license.

2.4 FatFs features

2.4.1 Duplicate file access

FatFs module does not support the sharing controls of duplicated file access in default. It is permitted when open method to a file is only read mode. The duplicated open in write mode to a file is always prohibited and open file must not be renamed, deleted, or the FAT structure on the volume can be collapsed.

The file sharing control can also be available when _FS_LOCK is set to 1 or greater. The value specifies the number of files to manage simultaneously. In this case, if any open, rename or remove that violates the file sharing rule that described above is attempted; the file function will fail with FR_LOCKED. If the number of open files gets larger than _FS_LOCK, the f_open() function will fail with FR_TOO_MANY_OPEN_FILES.

2.4.2 Reentrancy

The file operations to the different volumes are always reentrant and can work simultaneously. The file operations to the same volume are not reentrant but it can also be configured to thread-safe with _FS_REENTRANT option. In this case, also the OS dependent synchronization object control functions, ff_cre_syncobj(), ff_del_syncobj(), ff_req_grant() and ff_rel_grant() must be added to the project.

When a file function is called while the volume is in use by any other task, the file function is suspended until that task leaves file function. If the wait time exceeded a period defined by _TIMEOUT, the file function will abort with FR_TIMEOUT. The timeout feature might not be supported on some RTOS.

There is an exception on f_mount() and f_mkfs() functions. These functions are not reentrant to the same volume. When using these functions, all other tasks must close the corresponding file on the volume and avoid accessing the volume.

Note that this section describes the reentrancy of the FatFs module itself, but also the low level disk I/O layer must be reentrant.

2.4.3 Long file name

The FatFs module has started to support long file name (LFN) at revision 0.07. The two different file names, SFN and LFN, of a file is transparent in the file functions except for f_readdir() function. To enable LFN feature, set _USE_LFN to 1, 2 or 3, and add a Unicode code conversion function ff_convert() and ff_wtoupper() to the project. The LFN feature requires a certain working buffer in addition. The buffer size can be configured by _MAX_LFN corresponding to the available memory size. The size of long file name will reach up to 255 characters so that the _MAX_LFN should be set to 255 for full featured LFN operation. If the size of working buffer is insufficient for the given file name, the file function fails with FR_INVALID_NAME. When enabling the LFN feature with reentrant feature, _USE_LFN must be set to 2 or 3. In this case, the file function allocates the working buffer on the stack or heap. The working buffer occupies (_MAX_LFN + 1) * 2 bytes.

When the LFN feature is enabled, the module size will be increased depending on the selected code page. Right table shows how many bytes increased when LFN feature is enabled with some code pages.

2.5 FatFs APIs

The FatFs APIs layer implements file system APIs. It uses disk I/O interface to communicate with the appropriate physical drive. The set of APIs is divided into four groups:

- Group of APIs that operates with logical volume or partition.
- Group of APIs that operates with directory.
- Group of APIs that operates with file.
- Group of APIs that operates with both file and directory.

The following list describes what FatFs can do to access the FAT volumes:

- f_mount(): Register/Unregister a work area
- f_open(): Open/Create a file
- f_close(): Close a file
- f_read(): Read a file
- f_write(): Write a file
- f_lseek(): Move read/write pointer, Expand a file size
- f_truncate(): Truncate a file size
- f_sync(): Flush cached data
- f_opendir(): Open a directory
- f_readdir(): Read a directory item
- f_getfree(): Get free clusters
- f_stat(): Check if the object is exist and get status
- f_mkdir(): Create a directory

- f_unlink(): Remove a file or directory
- f_chmod(): Change an attribute
- f_utime(): Change timestamp
- f_rename(): Rename/Move a file or directory
- f_chdir(): Change the current directory
- f_chdrive(): Change the current drive
- f_getcwd(): Retrieve the current directory
- f_getlabel(): Get volume label
- f_setlabel(): Set volume label
- f_forward(): Forward file data to the stream directly
- f_mkfs(): Create a file system on the drive
- f_fdisk(): Devide a physical drive
- f_gets(): Read a string
- f_putc(): Write a character
- f_puts(): Write a string
- f_printf(): Write a formatted string
- f_tell(): Get the current read/write pointer
- f_eof(): Test for end-of-file on a file
- f_size(): Get the size of a file
- f_error(): Test for an error on a file

2.6 FatFs low level APIs

Since the FatFs module is completely separate from the disk I/O and RTC module, it requires some low level functions to operate the physical drive: read/write and get the current time. Because the low level disk I/O functions and RTC module are not a part of the FatFs module, they must be provided by the user.

The FatFs Middleware solution provides low level disk I/O drivers for some supported disk drives (RAMDisk, uSD, USBDisk).

An additional interface layer diskio.c has been added to add/remove dynamically (link) physical media to the FatFs module, providing low level disk I/O functions as mentioned below:

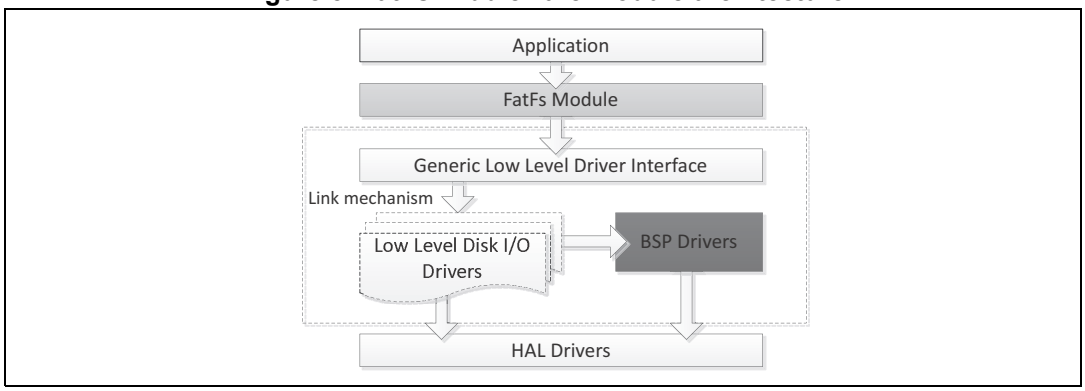
- disk_initialize(): Initializes the physical disk drive
- disk_status(): Returns the selected physical drive status
- disk_read(): Reads sector(s) from the disk
- disk_write(): Writes sector(s) to the disk
- disk_ioctl(): Controls device-specified features
- get_fattime(): Returns the current time

Application program MUST NOT call these functions, they are only called by FatFs file system functions such as, f_mount(), f_read(), f_write()…

2.7 FatFs into STM32CubeF4

In the STM32CubeF4 solution, an additional interface layer has been added to add/remove dynamically physical media to/from the FatFs module. To link FatFs module with a low level disk I/O driver, user can use FATFS_LinkDriver() and FATFS_UnLinkDriver() to add or remove dynamically a disk I/O driver; the application may need to know the number of current attached disk I/O drivers, this is done through the FATFS_GetAttachedDriversNbr() API.

Figure 5. FatFs Middleware module architecture



The generic low level driver ff_gen_drv.c/h is located in the root directory of the FatFs modules. Two disk I/O driver type definition structures are used to help dynamic management of attached disk drives under the ff_gen_drv.h file, as mentioned below:

Table 2. "Diskio_drv_TypeDef" structure

Field	Description
`DSTATUS (*disk_initialize)(void);`	Initialize Disk Drive
`DSTATUS (*disk_status)(void);`	Get Disk Status
`DRESULT (*disk_read)(BYTE*, DWORD, BYTE);`	Read Sector(s)
`DRESULT (*disk_write)(const BYTE*, DWORD, BYTE);`	Write Sector(s) _USE_WRITE should be = 0
`DRESULT (*disk_ioctl)(BYTE, void*);`	I/O control operation _USE_IOCTL should be = 1

Table 3. "Disk_drv_TypeDef" structure

Field	Description
`Diskio_drvTypeDef *drv[_VOLUMES];`	Diskio_drv_TypeDef structure
`uint8_t nbr;`	Number of the attached drives

To link FatFs module with a low level disk I/O driver, user can use the following APIs:

- FATFS_LinkDriver(): to add dynamically a disk I/O driver,
- FATFS_UnLinkDriver(): to remove dynamically a disk I/O driver,
- FATFS_GetAttachedDriversNbr(): to know the number of current attached disk I/O drivers

2.7.1 FATFS_LinkDriver()

This function links a compatible disk I/O driver and increments the number of active linked drivers. It returns 0 in case of success, otherwise it returns 1.

Note: Due to FatFs limits the MAX number of attached disks (_VOLUMES) is up to 10

Implementation of FATFS_LinkDriver:

```
uint8_t FATFS_LinkDriver(Diskio_drvTypeDef *drv, char *path)
{
  uint8_t ret = 1;
  uint8_t DiskNum = 0;
  if(disk.nbr <= _VOLUMES)
  {
    disk.drv[disk.nbr] = drv;
    DiskNum = disk.nbr++;
    path[0] = DiskNum + '0';
    path[1] = ':';
    path[2] = '/';
    path[3] = 0;
    ret = 0;
  }
  return ret;
}
```

2.7.2 FATFS_UnlinkDriver()

This function unlinks a disk I/O driver and decrements the number of active linked drivers. It returns 0 in case of success, otherwise it returns 1.

Implementation of FATFS_UnLinkDriver:

```
uint8_t FATFS_UnLinkDriver(char *path)
{
  uint8_t DiskNum = 0;
  uint8_t ret = 1;

  if(disk.nbr >= 1)
  {
    DiskNum = path[0] - '0';
    if(DiskNum <= disk.nbr)
    {
```

```
      disk.drv[disk.nbr--] = 0;
      ret = 0;
    }
  }

  return ret;
}
```

2.7.3 FATFS_GetAttachedDriverNbr()

This function returns the number of linked drivers to the FatFs module.

Implementation of FATFS_GetAttachedDriversNbr:

```
uint8_t FATFS_GetAttachedDriversNbr(void)
{
  return disk.nbr;
}
```

2.8 Interface your own disk to FatFs

If a working storage control module is available, it should be attached to the FatFs via a glue function rather than modifying it. User can interface any new disk by developing the appropriate disk I/O low level driver (mynewdisk_diskio.c/.h), and save these driver files under: \Middlewares\Third_Party\FatFs\src\drivers.

It is worth noting that the provided FatFs disk I/O low level drivers are dependent on the board BSP drivers. To remove this BSP dependency the user can just replace “BSP_...” APIs' calls by his own code ensuring the appropriate functionality.

To develop a disk I/O low level driver from scratch, the user can start from the skeleton of glue functions below to attach the existing storage control module to the FatFs with a defined API.

Low level disk I/O module skeleton for FatFs:

```
/*-----------------------------------------------------------------------*/
/* mynewdisk_diskio.c: Low level disk I/O module skeleton for FatFs      */
/*-----------------------------------------------------------------------*/

/* Includes ------------------------------------------------------------*/
#include <string.h>
#include "ff_gen_drv.h"
/* Private define ------------------------------------------------------*/
#define BLOCK_SIZE 512 /* Block Size in Bytes */

/* Private variables ---------------------------------------------------*/
static volatile DSTATUS Stat = STA_NOINIT; /* Disk status */

/* Private function prototypes -----------------------------------------*/
DSTATUS mynewdisk_initialize (void);
```

```
DSTATUS mynewdisk_status (void);
DRESULT mynewdisk_read (BYTE*, DWORD, BYTE);
#if _USE_WRITE == 1
  DRESULT mynewdisk_write (const BYTE*, DWORD, BYTE);
#endif /* _USE_WRITE == 1 */
#if _USE_IOCTL == 1
  DRESULT mynewdisk_ioctl (BYTE, void*);
#endif  /* _USE_IOCTL == 1 */

Diskio_drvTypeDef  mynewdisk_Driver =
{
  mynewdisk_initialize,
  mynewdisk_status,
  mynewdisk_read,
#if  _USE_WRITE == 1
  mynewdisk_write,
#endif /* _USE_WRITE == 1 */

/*------------------------ Initialize a Drive --------------------------*/
DSTATUS mynewdisk_initialize (void)
{
  Stat = STA_NOINIT;

  // write your own code here to initialize the drive

  Stat &= ~STA_NOINIT;
  return Stat;
}

/*------------------------ Get Disk Status -----------------------------*/
DSTATUS mynewdisk_status (void)
{
  Stat = STA_NOINIT;

  // write your own code here

  return Stat;
}

/*------------------------- Read Sector(s) -----------------------------*/
DRESULT mynewdisk_read (BYTE *buff, /* Data buffer to store read data */
                  DWORD sector, /* Sector address (LBA) */
                  BYTE count) /* Number of sectors to read (1..128) */
{
  DRESULT res = RES_ERROR;

  // write your own code here to read sectors from the drive

  return res;
}

/*------------------------- Write Sector(s) ----------------------------*/
#if _USE_WRITE == 1
DRESULT mynewdisk_write (const BYTE *buff, /* Data to be written */
```

```
                    DWORD sector, /* Sector address (LBA) */
                    BYTE count) /* Number of sectors to write (1..128) */
{
  DRESULT res = RES_ERROR;

  // write your own code here to write sectors to the drive

  return res;
}
#endif /* _USE_WRITE == 1 */

/*----------------------- Miscellaneous Functions -----------------------*/
#if _USE_IOCTL == 1
DRESULT mynewdisk_ioctl (BYTE cmd, /* Control code */
                    void *buff) /* Buffer to send/receive control data */
{
  DRESULT res = RES_ERROR;

  // write your own code here to control the drive specified features
  // CTRL_SYNC, GET_SECTOR_SIZE, GET_SECTOR_COUNT, GET_BLOCK_SIZE

  return res;
}
#endif /* _USE_IOCTL == 1 */
```

Header Low level disk I/O module:

```
/*-------------------------------------------------------------------------*/
/* mynewdisk_diskio.h: Header for Low level disk I/O module                */
/*-------------------------------------------------------------------------*/

/* Define to prevent recursive inclusion -------------------------------------*/
#ifndef __MYNEWDISK_DISKIO_H
#define __MYNEWDISK_DISKIO_H

extern Diskio_drvTypeDef  mynewdisk_Driver;

#endif /* __MYNEWDISK_DISKIO_H */
```

3 FatFs applications

In the STM32CubeF4 solution, many applications are provided based on FatFs middleware. The table below gives you insight on how the FatFs middleware component is used in different examples which are classified by complexity and depending on used physical drive interfaced (uSD, RAMDisk, USBDisk):

Table 4. Examples of FatFs middleware utilization

Example class	Examples	Description
Getting started	FatFs on single Logical Unit (RAMDisk)	Link FatFs module to a dummy disk I/O driver in RAM and perform mount, open, write, read, Close operation through a static buffer.
Features	FatFs on single Logical Unit	Link FatFs module to a uSD disk I/O driver and perform mount, open, write, read, close operations through a static buffer.
	FatFs on Multi Logical Unit	Link FatFs module to uSD and RAM disk I/O driver and perform mount, open, write, read, close operations through a static buffer.
Integrated	FatFs on single Logical Unit (USB Disk)	Link FatFs module to USB Host diskI/O driver and perform mount, open, write, read, close operations through a static buffer.

The FatFs applications listed above provided within STM32CubeF4 solution are a set of firmware available in two modes:

- Standalone mode
- RTOS mode, using *FreeRTOS* middleware component.

It is worth noting that user must guarantee appropriate values of stack and heap, when using or developing FatFs applications based on ST provided disk I/O low level drivers.

Thus, stack value must be incremented by the handled maximum sector size _MAX_SS value, available within ff_conf.h file, when using USB Disk application based on USB Host Mass Storage Class (MSC) for scratch alignment reasons.

Heap value must be also adjusted when developing any FatFs application in RTOS mode, using FreeRTOS middleware component based on CMSIS-OS wrapping layer common APIs.

3.1 HAL drivers configuration

FatFs applications provided within STM32CubeF4 solution is a set of firmware used to interface different physical disk drives (uSD, RAM Disk, USB Disk). User needs some HAL drivers which are essential to run the FatFs application. The correspond HAL drivers are enabled through the HAL configuration file *stm32f4xx_hal_conf.h*, by uncommenting the right modules used in the HAL driver.

The main difference in HAL configuration files, between the supported disk drivers is the definition of the right HAL driver corresponding to the used disk drive. The following defines must be available depending on each drive:

- FatFs_uSD:
  - #define HAL_SD_MODULE_ENABLED
- FatFs_RAMDisk:
  - #define HAL_SDRAM_MODULE_ENABLED or
  - #define HAL_SRAM_MODULE_ENABLED
- FatFs_USBDisk:
  - #define HAL_HCD_MODULE_ENABLED

3.2 FatFs File System configuration

FatFs module contains various configuration options. At this level we provide information to help user select proper options depending on the interfaced physical disk drives his requirement to reach the highest performance.

3.2.1 Reentrancy

Reentrancy is the key difference between the Standalone and the RTOS modes' configurations, which can be set on FatFs configuration file *ffconf.h*:

- Reentrancy is disabled in Standalone mode:
  - #define _FS_REENTRANT 0
- Reentrancy is enabled in RTOS mode:
  - #define _FS_REENTRANT 1

Once enabled, user must provide the OS dependent type of synchronization object (#define _SYNC_t osSemaphoreId)

RTOS mode applications' projects need to include the *syscall.c* file providing the OS depending functions, and found under: *\Middlewares\Third_Party\FatFs\src\option*

3.2.2 Long file name

The FatFs module supports long file name (LFN) and 8.3 format file name (SFN).

Note that the LFN feature on the FAT file system is a patent of Microsoft Corporation. This is not the case on FAT32 but most FAT32 drivers include the LFN feature. FatFs can switch the LFN feature by configuration option. When enable LFN feature on the commercial products, a license from Microsoft may be required depends on the final destination. The LFN can be used when LFN feature is enabled, which can be set on FatFs configuration file *ffconf.h*: (_USE_LFN > 0) within FatFs configuration file *ffconf.h*:

- LFN feature is disabled:
  - #define _USE_LFN 0
- LFN feature is enabled, when 3 ≥ _USE_LFN > 0:

Once enabled on *ffconf.h* configuration file, the application project needs to include the *syscall.c/unicode.c* files providing memory management functions, and found under: *\Middlewares\Third_Party\FatFs\src\option*

User can enable LFN feature either on standalone mode applications or in RTOS mode ones.

3.3 FatFs sample application

If user has already attached its own disk, developing the appropriate disk I/O low level driver (mynewdisk_diskio.c/.h), refer to *Section 2.8: Interface your own disk to FatFs*, linking this driver to FatFs module and using its logical disk can be done as follows:

```
/*-----------------------------------------------------------------------*/
/* main.c: Main program body                                             */
/*-----------------------------------------------------------------------*/

/* Includes ------------------------------------------------------------*/
#include "main.h"

/* Private variables ---------------------------------------------------*/
FATFS mynewdiskFatFs;  /* File system object for User logical drive */
FIL MyFile;            /* File object */
char mynewdiskPath[4]; /* User logical drive path */

int main(void)
{
  uint32_t wbytes; /* File write counts */
  uint8_t wtext[] = "text to write logical disk"; /* File write buffer */

if(FATFS_LinkDriver(&mynewdisk_Driver, mynewdiskPath) == 0)
  {
    if(f_mount(&mynewdiskFatFs, (TCHAR const*)mynewdiskPath, 0) == FR_OK)
    {
      if(f_open(&MyFile, "STM32.TXT", FA_CREATE_ALWAYS | FA_WRITE) == FR_OK)
      {
        if(f_write(&MyFile, wtext, sizeof(wtext), (void *)&wbytes) == FR_OK);
        {
          f_close(&MyFile);
        }
      }
    }
  }
  FATFS_UnLinkDriver(mynewdiskPath);
}
```

User must include the generic drive, *ff_gen_drv.h*, header file and also the disk IO module header file, *mynewdisk_diskio.h*

```
/*-----------------------------------------------------------------------*/
/* main.h: Header for main.c module                                      */
/*-----------------------------------------------------------------------*/
```

```
/* Includes ------------------------------------------------------------------*/
#include "ff_gen_drv.h"
#include "mynewdisk_diskio.h"
```

4 Conclusions

This User Manual explains how to integrate the FatFs middleware components within the STM32Cube HAL drivers.

A set of examples have been described to help users who develop applications based on FatFs File System within STM32Cube solution.

5 FAQ

How to use LFN feature with FatFs?

The FatFs module supports long file name (LFN). For more details on how to use LFN feature with FatFs refer to *Section 2.4.3: Long file name* and *Section 3.2.2: Long file name*.

What's the difference between FatFs Multi-partitions and Multi-drives applications?

Multi-partitions application can use multi logical drivers that can be bound to partitions on the specified physical drive, otherwise Multi-drives applications use different logical drives at the same time (uSD, RAMDisk…). User can choose the number of logical drives (volumes) to be used through _VOLUMES definition within FatFs *ffconf.h* configuration file.

Can user interface any new Disk to FatFs?

Yes, the user can interface a new disk to FatFs. For more details, refer to *Section 2.8: Interface your own disk to FatFs*.

Does FatFs support Multi-instances?

No, it doesn't. STM32CubeF4 solution provides the multi instance feature for HAL drivers, but, concerning FatFs middleware component, it cannot really support physical drivers multi-instances. In other words, the user can't hold an application using more than one instance of a logical drive.

What FAT sub-types does FatFs support?

FatFs refers to all three major variants of Microsoft FAT sub-types: FAT12, FAT16 and FAT32. The FAT sub-type is determined by number of clusters on the volume and nothing else, according to the FAT specification issued by Microsoft. Thus which FAT sub-type is selected, is depends on the volume size and the specified cluster size.

6 Revision history

Table 5. Document revision history

Date	Revision	Changes
04-Mar-2014	1	Initial release.
23-Jun-2014	2	On cover page updated: – document title – reference at STM32CubeF4 into STM32Cube

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com