CMX Systems, Inc.

SAFE File System

Implementation Guide

Version 2.40

Contents

1. System Overview

1.1. Summary

SAFE is a package of source code and documentation designed for flash file system development in embedded systems.

The following are the major features of the system:

General
- ANSI C compliant source code
- Extremely robust - guaranteed to be 100% safe against power-failure
- Syntax checked
- Easy-to-understand structure
- Scalable
- Easy portability to any development environment
- Minimal requirements from the host system

API
- Standard API
- Multi-user interface
- Long filenames
- Unicode 16 support

NOR Flash Support
- Wear-leveling
- Bad block handling
- Easy porting for all known device types
- Sample driver with porting description

Atmel DataFlash Support
- Wear-leveling
- All devices supported
- Manages the 10K writes/sector limitation
- Failsafe implementation of DataFlash interface

NAND Flash Support
- Wear-leveling
- Bad-block management
- ECC algorithm
- Easy porting for all known device types
- Sample driver with porting description

Other Media Types
- Compact Flash, MMC, SD, SDHC cards

1.2. Target Audience

This guide is intended for use by embedded software engineers who have a knowledge of the C programming language and standard file API's, and who wish to implement a file system in any combination of RAM, NAND flash, NOR flash and Atmel DataFlash.

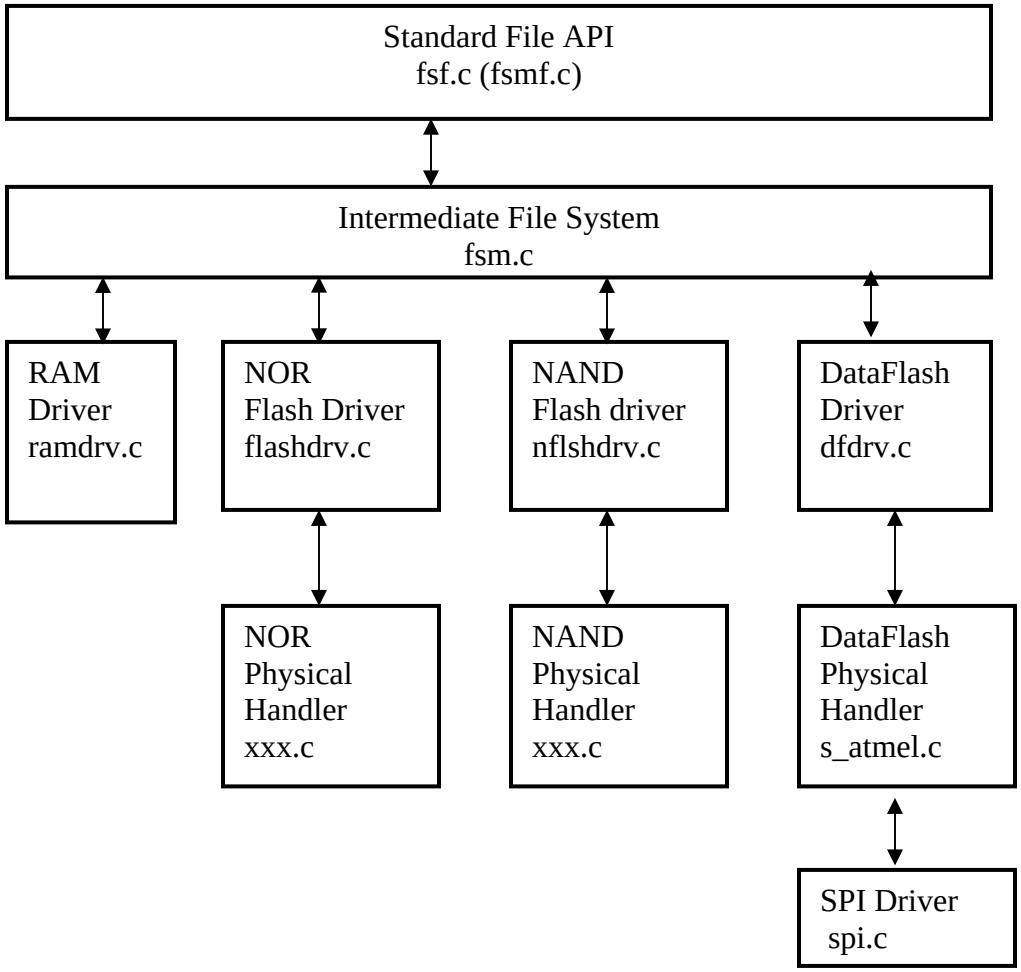
CMX Systems offers hardware and firmware development consultancy to assist developers with the implementation of flash file systems.

1.3. Important Notice

The SAFE file system was previously known as EFFS-STD. All references to STD in the code are historical and refer to the file system's original name.

1.4. System Structure/Source Code

The following diagram illustrates the structure of the file system software.



1.5. System Source File List

The following is a list of all the files included in the file system:

/src/common/

defs.h	- file system definitions file
fsf.c	- SAFE Standard API
fsf.h	- SAFE Standard API header
fsmf.c	- SAFE Standard API multi-thread wrapper
fsmf.h	- SAFE Standard API multi-thread wrapper header
fsm.c	- SAFE intermediate layer
fsm.h	- SAFE intermediate layer header
fstaticw.c	- static wear leveling functions
fstaticw.h	- static wear leveling header file
fwerr.h	- error code definitions
port_s.c	- functions to be ported
port_s.h	- header file for port functions
udefs.h	- user definitions header file
api_s.h	- provides all the definitions required to use the file API

File **api_s.h** is intended for use when the library is provided in object form under special license.

/src/test/

test.c	- test program source for exercising the file system
test.h	- header file for test program
testdrv.c	- test program source for exercising a flash driver
testdrv.h	- header file for flash driver test program
testport_ram_s.c	- sample port file for running test applications

/src/ram/

ramdrv_s.c	- RAM driver
ramdrv_s.h	- RAM driver header

/src/nor/ **Included only with SAFE-DF option**

flashdrv.c	- NOR flash driver
flashdrv.h	- NOR flash driver header

/src/nor/phy/amd/

29lvxxx.c	- NOR flash physical handler for AMD 29lxxx
29lvxxx.h	- NOR flash physical handler header

/src/nor/phy/atmel/

at49xxxx.c	- NOR flash physical handler for Atmel at49xxxx
at49xxxx.h	- NOR flash physical handler header

/src/nor/phy/intel/

28f320j3.c	- NOR flash physical handler for Intel StrataFlash
28f320j3.h	- NOR flash physical handler header
28f128j3pre.c	- NOR physical handler for Intel StrataFlash with pre-erase
28f128j3pre.h	- NOR physical handler header with pre-erase

/src/dflash/ **Included only with SAFE-DF option**

dfdrv.c	- DataFlash generic driver source
dfdrv.h	- DataFlash generic handler header

/src/dflash/phy/atmel/

s_atmel.c	- DataFlash physical handler
s_atmel.h	- DataFlash physical handler header
spi.c	- SPI communication sample driver
spi.h	- SPI communication header file

/src/nand/ **Included only with SAFE-NAND option**

nflshdrv.c	- NAND flash driver
nflshdrv.h	- NAND flash driver header

/src/nand/phy/micron/

mt29f2g08aab.c	- NAND flash physical handler
mt29f2g08aab.h	- NAND flash physical handler header

/src/nand/phy/samsung/

k9f2816xoc.c	- NAND flash physical handler
k9f2816x0c.h	- NAND flash physical handler header

/src/nand/phy/stmicro/

nand128w3a.c	- NAND flash physical handler
nand128w3a.h	- NAND flash physical handler header

Note: The source files are stored in this directory structure to clearly indicate the functionality of different modules. However, the code makes no assumptions about this; therefore, the developer may copy all relevant source files into a common directory.

Note: The physical driver files that are supplied may differ from the ones listed here.

1.6. What are NOR and NAND Flash?

SAFE has been designed to allow the easy integration of all standard flash devices. But what are these devices?

Flash devices have certain basic properties in common:

- They are designed for the non-volatile storage of code and data.
- To write to an area it must first be erased, which changes all erased bits to 1. Programming consists of changing 1s to 0s. To change a 0 to a 1 an erase operation must be performed.
- Flash devices are all divided into erase units (blocks). In order to erase any part a whole block must be erased.
- Data areas all wear out after a number of erase cycles. The guarantee for the number of successful erase cycles varies among the chip types. Therefore, it is important for any file system that uses flash to manage the wearing of the flash. This is done by avoiding the overuse of any one block.

There are two basic types of Flash chips generally available today. They have quite distinct physical characteristics and thus require quite different handling.

1.6.1. NOR Flash

NOR flash has been the cornerstone of non-volatile memory in embedded systems for many years. NOR has two basic characteristics: it stores data in a non-volatile way and it can be accessed directly from an address bus (“random access”) and thus can be used to run code.

NOR flash has some drawbacks. Firstly the erase/write time is very long; even if quite small mounts of data are written an erase may cause a delay of as much as 2 seconds. Careful design of the SAFE file system has ensured that this kind of behavior is minimized, but in certain cases it isn’t avoidable.

1.6.2. NAND/AND Flash

NAND flash (also AND) is a newer type of flash chip technology whose primary difference is:

- NAND can store approximately four times as much data as NOR technology for about the same price.
- NAND has much faster erase and write times, and thus is a superior choice for applications that require regular data storage.

There is a price to be paid for the improved performance:

- Data cannot be accessed via a standard address/data bus; instead commands must be sent to set the address and then the data can be read/written sequentially.
- Chips come from the factory with a number of bad blocks that can never be used.
- Bits may flip unexpectedly (don't panic! - see below).

Because of these complications NAND chips are designed with some additional features:

- Each block is divided into a number of read/write pages (typically 512, 2048 or 4096 bytes in size).
- Each page has an associated "spare" area that is used to store error correction and block management information. By using this area effectively the general performance and reliability of the devices is very high.

The NAND flash driver contains the necessary spare area management and fast ECC algorithm.

1.6.3. NOR/NAND Summary

The following table summarizes the differences between NOR and NAND flash types. The given data are indicative and subject to change as the technologies evolve.

Property	NOR	NAND
Price	4x/MB	x/MB
Size	64KB - 64MB	16MB - 8GB
Bootable	Yes	No
Random Access	Yes	No
Guaranteed Erase Cycles	10,000 - 100,000	1,000,000 with ECC
Block Erase Time (1)	2 sec	2 ms
Write Time (2)	10 µs/word	200 µs/page
Read Time (2)	100 ns/word	50 µs/page

Table 1, NOR/NAND Summary

1. Blocks on NAND flash devices are normally smaller than blocks on NOR flash devices. Since an erase must precede writing, the smaller block size is generally beneficial to a file system's performance.

2. The page size for NAND flash devices is typically 512, 2048 or 4096 bytes. Because file system access to the physical device is only in sectors, the page access times are the most important factors that affect the performance of the file system.

> **Note:** New devices with new features are being produced all the time. The above table should be used as an indication. For any particular chip type the specific datasheet for that device must be consulted.

1.7. Other Media Types

The base file system is designed based on the concept of a storage device with a logical block arrangement. Because of this, any device which can emulate a logical block arrangement can be used as a storage media for SAFE.

CMX Systems have available drivers for Compact Flash and for SD cards which can be used to run an entirely failsafe file system on these media – where the "normal" thing to do is to run a FAT file system.

1.8. Multiple Tasks, Mutexes and Reentrancy

If your system has only a single task that accesses the file system, then no changes to **port_s.c** are required for re-entrancy.

If your system has multiple tasks then this section must be understood and implemented.

Each volume should be protected by a mutex mechanism to ensure that file access is safe. A reentrancy wrapper is included in **fsm.c**. The reentrancy wrapper routines call mutex routines contained in **port_s.c**. These are general functions and should be replaced by the routines provided by your operating system, if there is one.

> **Note**: The mutex routines supplied with the system are vulnerable to the classic priority inversion problem. This problem can be resolved only by using the routines that are specific to the target's RTOS.

If reentrancy is required then the following functions in **port_s.c** must be implemented. They are normally provided by the host RTOS:

> ***f_mutex_create()*** – called at volume initialization
> ***f_mutex_delete()*** – called at volume deletion
> ***f_mutex_get()*** – called when a mutex is required
> ***f_mutex_put()*** – called when the mutex is released

> **Note**: If the Common API (CAPI) is used (i.e., F_CAPI_USED is defined in **udefs_s.h**), then these mutex functions will be replaced by those of the CAPI. Consult the CAPI guide for further information.

Within the standard API there is no support for the current working directory to be maintained on a per-caller basis. By default the system provides a single **cwd** that can be changed by any user. The **cwd** is maintained on a per-volume basis or on a per-task basis if multitasking is implemented.

For a multitasking system the developer must take the following steps:

1. Set F_MAXTASK to the maximum number of tasks that can simultaneously maintain access to the file system. This effectively creates a table of **cwd**s for each task.
2. Modify the function ***fn_gettaskID()*** in the **port_s.c** file to get a unique identifier for the calling task.
3. Ensure that any task using the file system calls ***f_enterFS()*** before using any other API calls; this ensures that the calling task is registered.

4. Ensure that for any application that has finished using the file system or is terminated, ***f_releaseFS()*** is called with the tasks unique identifier. This frees that table entry for use by other applications.

Once these steps are implemented each caller will be logged as it acquires the mutex, and a current working directory will be associated with it.

> **Note**: If the CAPI is used (i.e., F_CAPI_USED is defined in **udefs_s.h**) then ***fn_gettaskID()*** will be replaced by that in the CAPI. Consult the CAPI guide for further information.

1.9. Implementing Drivers

The CMX SAFE file system driver design achieves a high level of portability while still maintaining excellent performance of the system. The basic device architecture includes a high level driver for each general media type that shares some common properties. This driver handles issues of FAT maintenance, wear-leveling, etc. Below this lies a physical device handler that does the translation between the driver and the physical flash hardware.

Later sections of this manual contain a detailed description of the implementation for NOR and NAND flash.

Generally only the physical handler needs to be modified when the hardware configuration changes (e.g., different chip type, 1/2/4 devices in parallel, etc). CMX Systems has a range of physical handlers available to make the porting process as simple as possible. CMX Systems also offers special porting services when required.

1.10. System Requirements

The SAFE system is designed to be as open and portable as possible. No assumptions are made about the functionality or behavior of the underlying operating system. For SAFE to work at its best, certain porting work should be done as outlined below. This is a very straightforward task for an experienced engineer.

1.10.1. Timeouts

Flash devices are normally controlled by hardware control signals. As a result there is no explicit need for any timeouts to control exception conditions. However, some operations on flash devices are relatively slow and it is often worthwhile to schedule other operations while waiting for them to complete (e.g., a NOR flash erase typically takes 2 seconds and a NAND flash erase takes 2 milliseconds).

For NOR flash in the **29lvxxx.c** sample driver the ***DataPoll*** function is used to check for the completion of operations. This routine could be modified to force scheduling of the system or to use the event generation mechanism of the host system so that other operations can be performed while waiting.

For NAND flash in the K9F2816X0C sample driver the ***nandwaitrb*** function is used to check for the completion of operations. This routine could be modified to force scheduling of the system or to use the event generation mechanism of the host system so that other operations can be performed while waiting.

1.10.2. Real Time Clock

Whenever a file is created or closed (for writing) the system sets a date/time field associated with the file. To do this the following function in **port_s.c** is called:

```
void fs_getcurrenttimedate(
               unsigned short *ptime,
               unsigned short *pdate)
```

This function by default enters zeroes into these fields. When porting to a system with a real-time clock, the function should be modified to set the correct current time and date from your system. A recommended format for how this can be done is given by the following shift and mask definitions in the **fsm.h** file:

```
/* definitions for time */

#define FS_CTIME_SEC_SH 0
#define FS_CTIME_SEC_MASK     0x001f            /*     0-30     in
2seconds */
#define FS_CTIME_MIN_SH 5
#define FS_CTIME_MIN_MASK     0x07e0            /*   0-59   minutes
*/
#define FS_CTIME_HOUR_SH   11
#define FS_CTIME_HOUR_MASK    0xf800            /* 0-23 hours */

/* definition for date */

#define FS_CDATE_DAY_SH    0
#define FS_CDATE_DAY_MASK     0x001f      /* 0-31 days */
#define FS_CDATE_MONTH_SH     5
#define FS_CDATE_MONTH_MASK   0x01e0      /* 1-12 months */
#define FS_CDATE_YEAR_SH      9
#define FS_CDATE_YEAR_MASK    0xfe00      /*      0-119      (year
1980+value) */
```

N**ote**: Although this format is recommended, the developer may use these two 16 bit fields as they require - they will simply be updated according to the developer's replacement function each time a file is created or closed.

Note: If the CAPI is used (i.e., F_CAPI_USED is defined in **udefs_s.h**) then ***fs_getcurrenttimedate()*** will be replaced by that in the CAPI. Consult the CAPI guide for further information.

1.10.3. Memory Allocation

There are some larger buffers required by the file system to handle FATs in RAM and also to buffer write processes.

There is a call to each driver to get the specific size of memory required for that drive. It is then up to the user to allocate this memory from the system.

Buffer sizes depend on the particular chips being used and their configurations. For further information see the descriptions of the ***f_mountdrive*** and ***fs_getmem_xxx*** functions in the relevant driver sections.

1.10.4. Stack Requirements

SAFE file system functions are always called in the context of the calling thread or task. Naturally the functions require stack space and the developer should allow for this in applications that call file system functions. Typically calls to the file system will use <2KBytes of stack.

1.10.5. Memcpy and Memset

The SAFE system includes ***memcpy*** and ***memset*** functions, which are provided as simple byte copy routines. To get best performance from the target platform these routines should be replaced with routines developed specifically for the target system. As in all embedded systems the copy routines are time consuming but optimized versions can yield excellent performance benefits.

1.11. System Features

1.11.1. Power Fail Safety

The flash file system is entirely safe against power failure. The system may be stopped at any point and then restarted, and no data will be lost; the previously completed state of the file system will be restored.

When a file is closed its data are automatically flushed from the file system. Until this close takes place the file is preserved. The user may also use the ***f_flush*** command to write the current state of the file to the medium and thus update its failsafe state.

1.11.2. Long Filenames

The SAFE file system supports file names of almost unlimited length. Filename handling is efficient. It is built from a chain of small fragments taken from the descriptor block. If a filename is longer than FS_MAXDENAME (default 13), an additional FS_MAXLFN (default 11) byte block is allocated to store the longer name. These additional blocks are added by the file system automatically.

In **fsm.h** there is a FS_MAXLNAME define which sets the maximum allowed name length. By default this is set to FS_MAXDENAME+4*FS_MAXLFN (57 bytes). The developer may increase (or decrease) this by multiples of FS_MAXLFN bytes by changing the multiplier of FS_MAXLFN in the FS_MAXLNAME definition. This sets the number of these structures that may be used for a single name.

Long filenames use memory from the descriptor blocks in the file system. The system uses an efficient algorithm for allocating additional blocks in units of FS_MAXLFN. However, the use of long filenames reduces the number of file and directory entries that can be stored.

1.11.3. Multiple Volumes

The SAFE file system supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

The maximum number of volumes allowed by your system should be set in the FS_MAXVOLUME definition in **udefs.h**. Set this value to the maximum volume number used. If only a RAM drive is used, set the value to 1; if you use a RAM drive and NOR flash, then set this value to 2, etc. Volume letters are assigned by passing a parameter in the ***f_mountdrive*** function.

1.11.4. Multiple Open Files in a Volume

SAFE allows multiple files to be opened simultaneously on a volume or on different volumes. Within each driver (**ramdrv_s.c**, **flashdrv.c**, **nflshdrv.c**) there is a MAXFILE definition that determines the number of files that are allowed to be opened simultaneously on that volume at any particular time.

For each opened file an array must be allocated that contains a sector size buffer. Thus, increasing MAXFILES for a particular volume increases the RAM required by the system.

1.11.5. Case Sensitive File Names

By default, SAFE uses case insensitive names. To enable case sensitive names set FS_EFFS_CASE_SENSITIVE in **udefs.h** to 1.

1.11.6. Separator Character

You can define the file separator character using the FS_SEPARATORCHAR definition in **udefs.h**. By default this is a backslash.

1.11.7. Unicode

To enable the use of API functions for Unicode 16, convert the comment "#define CMX_UNICODE" in **udefs.h** to an active statement from its default value as a comment. Consult the API sections for the functions with prefixed w's for usage information. These are used with Unicode. An example is ***f_wopen()*** instead of ***f_open()***.

1.11.8. CAPI Enabled

If you are using FAT in the same system as SAFE then you can use CAPI to provide a common API for accessing both systems. To enable this feature, you should set FS_CAPI_USED in **udefs.h** to 1. If you are using SAFE on its own you should not change this setting.

1.11.9. Static Wear Leveling

Flash devices are usually manufactured to a specification that includes a guaranteed number of write-erase cycles that can be performed on each block before it may develop a fault. Because of this it is important to use the blocks in a device "evenly" if the device is to be used for its maximum lifetime.

SAFE uses a process called dynamic wear leveling to allocate the least-used blocks from the available ones. However, in systems where there are large areas of static data (e.g., the executable binary for the system), the areas may be written only once, thus leaving a relatively small section of the device to handle the much more heavily used files.

For this reason a process called static wear leveling is introduced. When the ***fs_staticwear*** function is called it searches for blocks that have been used much less than the most used blocks in the system. If this difference is greater than a defined threshold (FS_STATIC_DISTANCE), then these two blocks will be exchanged in the system.

The files **fstaticw.c** and **fstaticw.h** must be included in your project in order to use the static wear leveling feature. Two defines should be set in the header file:

FS_STATIC_DISTANCE – This specifies the minimum difference between a heavily used block and a lightly used block before a static swap is allowed. This number should not be so small that it causes unnecessary swapping. A reasonable figure is between 1% and 10% of the guaranteed erase/write cycles of the target chip.

FS_STATIC_PERIOD – This specifies how often this function will actually attempt to do a swap. It may be used to reduce the number of times that ***fs_staticwear*** will be executed. This reduces unnecessary checking of the system. If you always know that the system will be idle when ***fs_staticwear*** is called then this may be set to 1 so that it is always executed; for example, if you just do a few calls to ***fs_staticwear*** at start-up. If it is called at every available opportunity then you may want to execute it less frequently.

When the static wear leveling function is executing, the file system is not accessible. The length of time it takes depends on the specification of the target chips being used and in particular the time required to erase a block and the time required to copy one block to another.

For static wear leveling to function, an additional driver function, ***BlockCopy***, must also be provided. See the appropriate driver sections in this manual (3.4.6 for NOR flash or 5.5.9 for NAND flash) for information as to how to implement this function for your target media. It is important to provide a highly optimized version of ***BlockCopy***, preferably by using special copy functions that are specific to the target chip, in order to achieve the best system performance and least system disruption.

Do I need static wear-leveling?

In many cases it is unnecessary overhead. To assess its importance look at how your product is to be used and consider the specifications of your target devices. Many devices have up to 1 million guaranteed erase/write cycles per block and in many applications this number will not be reached in the lifetime of the product.

When should I do static wear-leveling?

Because wear-leveling involves swapping blocks in the file system, all access is excluded for the duration of the process. Thus, if there are time-critical features in your device then it is preferable to do static wear-leveling during idle moments. A possible time is during system boot where several static wear-levels could be done without having a major impact on the boot time of the system.

1.11.10. Free Block Allocation

The system includes two different algorithms for allocating file system blocks. One just finds a block with a single available sector; the second (default) allocates the block which has the most free sectors.

The algorithm choice is made with the FSF_MOST_FREE_ALLOC define in **fsf.h**. The default setting is recommended.

1.12. Getting Started

To get your development started as efficiently as possible, take the following steps:

1. Build the file system using the API (**fsf.c, fsmf.c**), the intermediate file system (**fsm.c**) and ported functions (**port_s.c**) and the RAM driver (**ramdrv_s.c**), including the relevant header files. In this way you can build a file system that runs in RAM with little or no dependency on your hardware platform.
2. Build a test program to exercise this file system and check how it works in RAM. All build and integration issues can thus be addressed before worrying about specific flash devices.
3. Now add the next volume to the system: either a NOR drive or a NAND drive, depending on your requirements.
   For NOR drive:
   Add **flashdrv.c** to the build.
   For DataFlash drive:
   Add **dfdrv.c** to the build.
   For NAND drive:
   Add **nflshdrv.c** to the build.
4. Now add a physical device driver to the build.
   For NOR chips:
   Read Section 3 "NOR Flash Driver" carefully and, using the available sample drivers as a basis, create a driver that meets your specific needs.
   For DataFlash chips:
   Read Section 4 "Atmel DataFlash Driver" carefully and, using the available sample drivers as a basis, create a driver that meets your specific needs.
   For NAND chips:
   Read Section 5 "NAND Flash Driver" carefully and, using the available sample drivers as a basis, create a driver that meets your specific needs.
5. Add new volumes by repeating steps 3 and 4.

2. File System API

Common functions

f_getversion
f_releaseFS
f_getfreespace
f_get_oem
f_init
f_mountdrive
fs_staticwear
f_enterFS
f_format
f_unmountdrive

Drive\Directory handler functions

f_get_drive_list
f_chdrive
f_getdcwd
f_wmkdir
f_rmdir
f_setlabel
f_get_drive_count
f_getcwd
f_wgetdcwd
f_chdir
f_wrmdir
f_getlabel
f_getdrive
f_wgetcwd
f_mkdir
f_wchdir
f_checkvolume

File functions

f_rename
f_wmove
f_filelength
f_wfindfirst
f_wsettimedate
f_setpermission
f_wgetpermission
f_stat
f_wrename
f_delete
f_wfilelength
f_findnext
f_gettimedate
f_wsetpermission
f_truncate
f_wstat
f_move
f_wdelete
f_findfirst
f_settimedate
f_gettimedate
f_getpermission
f_wtruncate
f_ftruncate

Read/Write functions

f_open
f_flush
f_seek
f_putc
f_tell
f_wopen
f_write
f_eof
f_getc
f_close
f_read
f_rewind
f_seteof

2.1. f_getversion

This function is used to retrieve file system version information.

Format

```
char * f_getversion(void)
```

Arguments

None

Return values

Return value	Description
char *	pointer to null terminated ASCII string

Example:

```
void display_fs_version(void)
{
   printf("File System Version: %s",f_getversion());
}
```

2.2. f_init

This function initializes the file system. It must be called once, before using any other file system function.

This function sets all critical initialized data to zero. This feature is required only if your compiler does not set un-initialized static data to zero – a feature of TI's Code Composer.

Format

```
int f_init(void)
```

Arguments

None

Return values

Return value	Description
F_NO_ERROR	drive successfully initialized
else	see error codes

Example

```
void main(void)
{
   f_init(); /* init file system */
      .
      .
}
```

See also

```
f_mountdrive, f_format
```

2.3. f_enterFS

If the target system allows multiple tasks to use the file system, then this function must be called by a task before using any other file API functions. This function creates resources for the calling task in the file system and allocates a current working directory for that task.

The ***f_releaseFS()*** call must be made to release the task from the file system and free the allocated resource.

The correct operation of this function also requires that ***fn_gettaskID()*** in **port_s.c** has been ported to provide a unique identifier for each task.

If the system is a single task based system then this function also needs to be called after ***f_init()*** is called.

Format

```
int f_enterFS(void)
```

Arguments

None

Return values

Return value	Description
F_NO_ERROR	if success
other	see error codes

Example

```
void main()
{
   f_init(); /* initialize filesystem */
   f_enterFS(); /* allow a task access to filesystem */
      .
      .
      .
}
```

2.4. f_releaseFS

This function is called by the user to release a previously assigned unique task ID.

The unique task identifier is generated by ***fn_gettaskID()*** in **port_s.c**

This function must be called if a given task is released or no longer exist.

Format

```
void f_releaseFS(long ID)
```

Arguments

Argument	Description
ID	unique identifier for calling task

Return values

Return value	Description
none	

Example

```
void task_destructor()
{
   f_releaseFS(fn_gettaskID()); /* release current task ID */
   .
   .
   .
}
```

2.5. f_mountdrive

This function is used to mount and map a new drive. It must be called with five parameters:

drivenum

Number of the drive to be mounted, where 0 is drive 'A', 1 is drive 'B', etc. The maximum value of **drivenum** is set in FS_MAXVOLUME-1 in **fsm.h**.

buffer

This is a pointer for a buffer area to be used by the generic driver. Its size depends on the specific devices and configuration used.

For a RAM drive a buffer of the size required for the whole RAM file system should be allocated as shown in the example below.

For a NOR drive the generic NOR flash function ***fs_getmem_flashdrive*** must be called with a pointer to the ***get-physical*** function of the specific physical chip driver to be mounted (e.g., ***fs_phy_nor_29lvxxx***). This function then calculates and returns the amount of memory that must be allocated for the physical driver. The caller must then allocate the memory and pass its pointer and size to the ***f_mountdrive*** function. See example code below.

For a NAND drive the generic NAND flash function ***fs_getmem_nandflashdrive*** must be called with a pointer to the ***get-physical*** function of the specific physical chip driver to be mounted (e.g., ***fs_phy_nand_K9F2816X0C***). This function then calculates and returns the amount of memory that must be allocated for the physical driver. The caller must then allocate this amount of memory and pass its pointer and size to the ***f_mountdrive*** function. See example code below.

buffsize

This is the size of the allocated buffer that's passed to the mount function.

mountfunc

This is a pointer to the generic mount function for the specific media type.

mountfunc is a driver function that describes which drive needs to be mounted. This calls the physical driver function to be associated with it.

Standard examples are:

```
fs_mount_ramdrive       - for using drive as RAM drive
fs_mount_flashdrive     - for using drive as NOR flash drive
fs_mount_nandflashdrive - for using drive as NAND flash drive
```

phyfunc

This is a pointer to a physical driver function for the desired device that is called by the generic mount function to get information about how to use the device. For a RAM drive this function is NULL.

Standard examples are:

```
fs_phy_nor_sim              - for PC emulation of NOR physical
fs_phy_nor_29lvxxx          - for AMD flash
fs_phy_nand_sim             - for PC emulation of NAND physical
fs_phy_nand_ K9F2816X0C     - for Samsung NAND flash
```

Format

```
int  f_mountdrive(int  drivenum,  void  *buffer,  long  buffsize,
      FS_DRVMOUNT mountfunc, FS_PHYGETID phyfunc)
```

Arguments

Argument	Description
drivenum	number of drive to be mounted (0='A' etc.)
buffer	buffer pointer to be used by file system
buffsize	size of buffer
mountfunc	mount function for selected drive type
phyfunc	physical driver for specific chip type

Return values

Return value	Description
FS_VOL_OK	successfully mounted
FS_VOL_NOTMOUNT	not mounted
FS_VOL_NOTFORMATTED	drive is mounted but drive is not formatted
FS_VOL_NOMEMORY	not enough memory, drive is not mounted
FS_VOL_NOMORE	no more drive available (FS_MAXVOLUME)
FS_VOL_DRVERROR	mount driver error, not mounted

Example

```
/* this example shows how to mount Ramdrive,    */
/* FLASH drive and NANDFLASHdrive               */

char p0buffer[0x100000]; /* 1M */

void main(void)
{
   char *p1buffer, *p2buffer;
   long memsize;

   f_init();
   f_enterFS();

   f_mountdrive(0,p0buffer,sizeof(p0buffer),fs_mount_ramdrive,
0);
   /* Drive A will be RAM drive */

   memsize=fs_getmem_flashdrive(fs_phy_nor_29lvxxx);
   if (!memsize)
   {
      /* flash is not identified */
   }

   p1buffer=(char*)malloc(memsize);
   if (!p1buffer)
   {
      /* Not enough memory to allocate */
   }

   f_mountdrive(1,p1buffer,memsize,fs_mount_flashdrive,
      fs_phy_nor_29lvxxx);
      /* Drive B will be NOR flash drive, with */
      /* AMD physical driver */

   memsize=fs_getmem_nandflashdrive(fs_phy_nand_K9F2816X0C);
   if (!memsize)
   {
      /* nand flash is not identified, */
   }
   p2buffer=(char*)malloc(memsize);
   if (!p2buffer)
   {
      /* Not enough memory to allocate */
   }

f_mountdrive(2,p2buffer,memsize,fs_mount_nandflashdrive,
   fs_phy_nand_K9F2816X0C);
      /* Drive C will be NAND flash drive with */
      /* Samsung physical                      */
}
```

See also

```
f_init, f_format, f_unmountdrive
```

2.6. f_unmountdrive

This function is used to unmount an existing volume. Any open files on the media will be marked as closed so that subsequent API accesses to a previously opened file handle will return with an error.

This function works independently of the status of the hardware.

Format

```
int f_delvolume(int drivenum)
```

Arguments

Argument	Description
drivenum	drive to be deleted (0:A, 1:B...)

Return values

Return value	Description
F_NO_ERROR	drive successfully deleted
else	see error codes

Example:

```
void mydelfs(int num)
{
   int ret;

   /*Delete volume 1 */
   if(f_unmountdrive (num))
      printf("Unable to delete volume %d", num);
          .
          .
          .
}
```

See also

```
f_mountdrive
```

2.7. f_get_drive_count

This function returns the number of drives currently available to the user.

Format

```
int f_get_drive_count(void)
```

Arguments

Argument	Description
none	

Return values

Return value	Description
num	number of active volumes

Example

```
void mygetvols(void)
{
   printf("there are %d active drives\n",
      f_get_drive_count());
          .
          .
}
```

See also

```
f_get_drive_list
```

2.8. f_get_drive_list

This function returns a list of drives currently available to the user.

Format

```
int f_get_drive_list(int *buffer)
```

Arguments

Argument	Description
none	

Return values

Return value	Description
number	number of active volumes

Example:

```
void mygetvols(void)
{
   int i,j;
   int buffer[F_MAXVOLUME];

   i=f_get_drive_list(buffer);

   if (!i) printf ("no active drive found\n");

   for (j=0;j<i;j++)
   {
      printf("Drive %d is active\n", buffer[j]);
   }

}
```

See also

f_get_drive_count

2.9. f_format

Format a drive. All data on the drive will be destroyed, with the exception of the wear-leveling information on a FLASH device.

Format

```
int f_format(int drivenum)
```

Arguments

Argument	Description
drivenum	which drive needs to be formatted

Return values

Return value	Description
F_NO_ERROR	drive successfully formatted
F_ERR_INVALIDDRIVE	if drive does not exist
F_ERR_BUSY	if there is any file open
F_ERR_NOTFORMATTED	if drive cannot be formatted

Example

```
char buffer[0x30000];

void myinitfs(void)
{
   int ret;

   f_init();
   f_enterFS();

   ret=f_mountdrive(0,buffer,sizeof(buffer),fs_mount_flashdrive,
      fs_phy_nor_29lvxxx);
      /* Drive A will be NOR flash drive */

   if (ret==FS_VOL_OK) return; /* initialized */
   if (ret==FS_VOL_NOTFORMATTED)
   {
      ret=f_format(0); /* format drive A */
      if (ret==F_ERR_NOTERR) return; /* formatted */
   }
initializationfailed:
}
```

See also

```
f_init, f_mountdrive
```

2.10. f_getfreespace

This function fills a user-allocated structure with information about the usage of the specified volume. Four items of information about the drive are returned:

1. total size
2. free space
3. used space
4. bad space

Format

```
int f_getfreespace(int drvnum, F_SPACE *pSpace)
```

Arguments

Argument	Description
drivernum	number of drive
pSpace	pointer to user's free space structure

Return values

Return value	Description
F_NO_ERROR	success
else	see error codes

Example

```
void info(void)
{
   int ret;
   F_SPACE space;

   ret = f_getfreespace(f_getdrive(),&space);

   if(!ret)
   {
      printf("There are %d total bytes,\
         %d free bytes,\
         %d used bytes,\
         %d bad bytes.\n",
         space.total,space.free,\
         space.used,space.bad);
   }
   else
      printf("Error %d\n",ret);
}
```

2.11. f_setlabel

This function sets a volume label. The volume label should be a NULL terminated ASCII string with a maximum length of FS_MAXDENAME characters.

Format

```
int f_setlabel(int drivenum, const char *pLabel)
```

Arguments

Argument	Description
drivenum	drive number
pLabel	pointer to null terminated string to use

Return values

Return value	Description
F_NO_ERROR	success
else	see error codes

Example

```
void setlabel(void)
{
   int result = f_setlabel(f_getcurrdrive(),"DRIVE 1");

   if (result)
     printf("Error on Drive");
}
```

2.12. f_getlabel

This returns the label as a function value. The pointer passed for storage should be able to hold a character string of length FS_MAXDENAME.

Format

```
int f_getlabel(int drivenum, char *pLabel, long len)
```

Arguments

Argument	Description
drivenum	drive number
pLabel	pointer to copy label to
len	length of storage area

Return values

Return value	Description
F_NOERROR	success
else	see error codes

Example

```
void getlabel(void)
{
   char label[FS_MAXDENAME];
   int result;

   result = f_getlabel(f_getcurrdrive(),label,FS_MAXDENAME);

   if (result)
      printf("Error on Drive");
   else
      printf("Drive is %s",label);
}
```

2.13. fs_staticwear

This function is called to even the wear of blocks that are rarely used.

Read the “Static Wear” part of Section 1 of this manual for information about when and how to use this function.

Format

```
int fs_staticwear(int drvnum)
```

Arguments

Argument	Description
drvnum	number of target drive

Return values

Return value	Description
F_NO_ERROR	success
else	see error code

Example

```
void idle(void)
{
   int ret;

   /* try static wear on Drive A */

   ret = fs_staticwear(0);

   if(!ret)
   {
      printf("Static Wear Done\n”);
   }
   Else
   {
      printf(“Error in static wear!!\n”,ret);
   }
}
```

2.14. f_mkdir

Make a new directory.

Format

```
int f_mkdir(const char *dirname)
```

Arguments

Argument	Description
dirname	new directory name to create

Return values

Return value	Description
F_NO_ERROR	new directory name created successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_INVALIDDIR	invalid path
F_ERR_DUPLICATED	entry already exists
F_ERR_NOMOREENTRY	directory is full

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir("subfolder");   /* creating directory */
   f_mkdir("subfolder/sub1");
   f_mkdir("subfolder/sub2");
   f_mkdir("a:/subfolder/sub3"
      .
      .
}
```

See also

```
f_chdir, f_rmdir
```

2.15. f_wmkdir

Make a new directory with a Unicode 16 name.

Format

```
int f_wmkdir(const W_CHAR *dirname)
```

Arguments

Argument	Description
dirname	new Unicode 16 directory name to create

Return values

Return value	Description
F_NO_ERROR	new directory name created successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_INVALIDDIR	invalid path
F_ERR_DUPLICATED	entry already exists
F_ERR_NOMOREENTRY	directory is full

Example

```
void myfunc(void)
{
      .
      .
   f_wmkdir("subfolder");  /* creating directory */
   f_wmkdir("subfolder/sub1");
   f_wmkdir("subfolder/sub2");
   f_wmkdir("a:/subfolder/sub3"
      .
      .
}
```

See also

```
f_wchdir, f_wrmdir
```

2.16. f_chdir

Change current working directory. In a multitask system every task has its own current working directory.

Format

```
int f_chdir(const char *dirname)
```

Arguments

Argument	Description
dirname	new directory name to change

Return values

Return value	Description
F_NO_ERROR	directory has been changed successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	path not found

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir("subfolder");
   f_chdir("subfolder");   /* change directory */
   f_mkdir("sub2");
   f_chdir(".."); /* go to upward */
   f_chdir("subfolder/sub2");
   /* goto into sub2 dir */
      .
      .
}
```

See also

```
f_mkdir, f_rmdir, f_getcwd, f_getdcwd
```

2.17. f_wchdir

Change the current working directory with a Unicode 16 name.

Format

```
int f_wchdir(const W_CHAR *dirname)
```

Arguments

Argument	Description
dirname	new Unicode 16 directory name to change

Return values

Return value	Description
F_NO_ERROR	directory has been changed successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	path not found

Example

```
void myfunc(void)
{
      .
      .
   f_wmkdir("subfolder");
   f_wchdir("subfolder");  /* change directory */
   f_wmkdir("sub2");
   f_wchdir("..");   /* go to upward */
   f_wchdir("subfolder/sub2");
   /* goto into sub2 dir */
      .
      .
}
```

See also

```
f_wmkdir, f_wrmdir, f_wgetcwd, f_wgetdcwd
```

2.18. f_rmdir

Remove directory. Directory must be empty when it is removed; otherwise an error code is returned without removing.

Format

```
int f_rmdir(const char *dirname)
```

Arguments

Argument	Description
dirname	directory name to remove

Return values

Return value	Description
F_NO_ERROR	directory name is removed successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_NOTFOUND	directory not found
F_ERR_INVALIDDIR	directory name is not a directory
F_ERR_NOTEMPTY	directory not empty

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir("subfolder"); /* creating directories */
   f_mkdir("subfolder/sub1");
      .
      . doing some work
      .
   f_rmdir("subfolder/sub1");
   f_rmdir("subfolder"); /* removes directory */
      .
      .
}
```

See also

```
f_mkdir, f_chdir
```

2.19. f_wrmdir

Remove Unicode 16 directory. Directory must be empty when it is removed; otherwise an error code is returned without removing.

Format

```
int f_wrmdir(const W_CHAR *dirname)
```

Arguments

Argument	Description
dirname	Unicode 16 directory name to remove

Return values

Return value	Description
F_NO_ERROR	directory name is removed successfully
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_NOTFOUND	directory not found
F_ERR_INVALIDDIR	directory name is not a directory
F_ERR_NOTEMPTY	directory not empty

Example

```
void myfunc(void)
{
      .
      .
   f_wmkdir("subfolder"); /* creating directories */
   f_wmkdir("subfolder/sub1");
      .
      . doing some work
      .
   f_wrmdir("subfolder/sub1");
   f_wmdir("subfolder"); /* removes directory */
      .
      .
}
```

See also

```
f_wmkdir, f_wchdir
```

2.20. f_getdrive

Get current drive number

Format

```
int f_getdrive(void)
```

Arguments

none

Return values

Return value	Description
Current Drive	0-A, 1-B, 2-C, etc.

Example

```
void myfunc(void)
{
   int currentdrive;
      .
   currentdrive=f_getdrive();
      .
      .
}
```

See also

```
f_chdrive
```

2.21. f_chdrive

Change current drive.

Format

```
int f_chdrive(int drivenum)
```

Arguments

Argument	Description
drivenum	drive number to be current drive (0-A,1-B,2-C,…)

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	drive number is invalid

Example

```
void myfunc(void)
{
      .
      .
   f_chdrive(0); /* select drive A */
      .
      .
}
```

See also

```
f_getdrive
```

2.22. f_getcwd

Get current working folder on current drive.

Format

```
int f_getcwd(char *buffer, int maxlen )
```

Arguments

Argument	Description
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	Success
F_ERR_INVALIDDRIVE	current drive is invalid

Example

```
void myfunc(void)
{
   char buffer[F_MAXPATH];

   if (!f_getcwd(buffer, F_MAXPATH))
   {
      printf ("current directory is %s",buffer);
   }
   else
   {
      printf ("Drive Error")
   }
}
```

See also

```
f_chdir, f_getdcwd
```

2.23. f_wgetcwd

Get current working folder on current drive.

Format

```
int f_wgetcwd(W_CHAR *buffer, int maxlen )
```

Arguments

Argument	Description
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	current drive is invalid

Example

```
void myfunc(void)
{
   W_CHAR buffer[F_MAXPATH];

   if (!f_wgetcwd(buffer, F_MAXPATH))
   {
      wprintf ("current directory is %s",buffer);
   }
   else
   {
      wprintf ("Drive Error")
   }
}
```

See also

```
f_wchdir, f_wgetdcwd
```

2.24. f_getdcwd

Get current working folder on selected drive.

Format

```
int f_getdcwd(int drivenum, char *buffer, int maxlen )
```

Arguments

Argument	Description
drivenum	specify drive (0-A, 1-B, 2-C, etc.)
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	current drive is invalid

Example

```
void myfunc(int drivenum)
{
   char buffer[F_MAXPATH];

   if (!f_getdcwd(drivenum,buffer, F_MAXPATH))
   {
      printf ("current directory is %s",buffer);
      printf ("on drive %c",drivenum+'A');
   }
   else
   {
      printf ("Drive Error")
   }
}
```

See also

f_chdir, f_getcwd

2.25. f_wgetdcwd

Get current working folder on selected drive.

Format

```
int f_wgetdcwd(int drivenum, W_CHAR *buffer, int maxlen )
```

Arguments

Argument	Description
drivenum	specify drive (0-A, 1-B, 2-C, etc.)
buffer	where to store current working directory string
maxlen	length of the buffer

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	current drive is invalid

Example

```
void myfunc(int drivenum)
{
   W_CHAR buffer[F_MAXPATH];

   if (!f_wgetdcwd(drivenum,buffer, F_MAXPATH))
   {
      wprintf ("current directory is %s",buffer);
      wprintf ("on drive %c",drivenum+'A');
   }
   else
   {
      wprintf ("Drive Error")
   }
}
```

See also

f_wchdir, f_wgetcwd

2.26. f_rename

Rename a file or directory. This function has been obsoleted by ***f_move***.

Format

```
int f_rename(const char *filename, const char *newname)
```

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of file or directory

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found
F_ERR_BUSY	file is open for read or write
F_ERR_DUPLICATED	name already exists

Example

```
void myfunc(void)
{
      .
      .
   f_rename ("oldfile.txt","newfile.txt");
   f_rename ("A:/subdir/oldfile.txt","newfile.txt");
      .
      .
}
```

See also

```
f_mkdir, f_open, f_move
```

2.27. f_wrename

Rename a file or directory with a Unicode 16 name. This function has been obsoleted by ***f_wmove***.

Format

```
int f_rename(const W_CHAR *filename, const W_CHAR *newname)
```

Arguments

Argument	Description
filename	Unicode 16 file or directory name with/without path
newname	new Unicode 16 name of file or directory

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found
F_ERR_BUSY	file is open for read or write
F_ERR_DUPLICATED	name already exists

Example

```
void myfunc(void)
{
      .
      .
   f_wrename ("oldfile.txt","newfile.txt");
   f_wrename ("A:/dir/oldfile.txt","newfile.txt");
      .
      .
}
```

See also

```
f_wmkdir, f_wopen, f_wmove
```

2.28. f_move

Moves a file or directory; the original is lost. This function obsoletes ***f_rename()***. The source and target must be in the same volume.

Format

```
int f_move(const W_CHAR *filename, const char *wnewname)
```

Arguments

Argument	Description
filename	file or directory name with/without path
newname	new name of file or directory with/without path

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found
F_ERR_BUSY	file is open for read or write
F_ERR_DUPLICATED	name already exists

Example

```
void myfunc(void)
{
      .
      .
   f_move ("oldfile.txt","newfile.txt");
   f_move ("A:/subdir/oldfile.txt", "A:/newdir/oldfile.txt");
      .
      .
}
```

See also

```
f_mkdir, f_open, f_rename
```

2.29. f_wmove

Moves a file or directory with a Unicode 16 name. The original is lost. This function obsoletes ***f_wrename***. The source and target must be in the same volume.

Format

```
int f_wmove(const W_CHAR *filename, const W_CHAR *newname)
```

Arguments

Argument	Description
filename	Unicode 16 file or directory name with/without path
newname	new Unicode 16 name of file or directory

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found
F_ERR_BUSY	file is open for read or write
F_ERR_DUPLICATED	name already exists

Example

```
void myfunc(void)
{
      .
      .
   f_wmove ("oldfile.txt","newfile.txt");
   f_wmove ("A:/subdir/oldfile.txt","A:/newdir/oldfile.txt");
      .
      .
}
```

See also

```
f_wmkdir, f_wopen, f_wrename
```

2.30. f_delete

Delete a file.

Format

```
int f_delete(const char *filename)
```

Arguments

Argument	Description
filename	file name to be deleted, with/without path

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file not found
F_ERR_BUSY	file is open for read or write
F_ERR_INVALIDDIR	file name is a directory name

Example

```
void myfunc(void)
{
      .
      .
   f_delete ("oldfile.txt");
   f_delete ("A:/subdir/oldfile.txt");
      .
      .
}
```

See also

```
f_open
```

2.31. f_wdelete

Delete a file with a Unicode 16 name.

Format

```
int f_wdelete(const W_CHAR *filename)
```

Arguments

Argument	Description
filename	file name with/without path to be deleted

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	filename contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file not found
F_ERR_BUSY	file is open for read or write
F_ERR_INVALIDDIR	file name is a directory name

Example

```
void myfunc(void)
{
      .
      .
   f_wdelete ("oldfile.txt");
   f_wdelete ("A:/subdir/oldfile.txt");
      .
      .
}
```

See also

f_wopen

2.32. f_filelength

Get the length of a file.

Format

```
long f_filelength (char *filename)
```

Arguments

Argument	Description
filename	file name with or without path

Return values

Return value	Description
filelength	length of file, if zero length then file may not exist – check last error.

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");
   long size=f_filelength(filename);

   if (!file)
   {
      printf ("%s Cannot be opened!",filename);
      return 1;
   }

   if (size>buffsize)
   {
      printf ("Not enough memory!");
      return 2;
   }

   f_read(buffer,size,1,file);
   f_close(file);

   return 0;
}
```

See also

```
f_open
```

2.33. f_wfilelength

Get the length of a file with a Unicode 16 name.

Format

```
long f_wfilelength (W_CHAR *filename)
```

Arguments

Argument	Description
filename	Unicode 16 file name with or without path

Return values

Return value	Description
filelength	length of file, if zero length then file may not exist – check last error.

Example

```
int myreadfunc(W_CHAR *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_wopen(filename,"r");

   long size=f_wfilelength(filename);
   if (!file)
   {
      printf ("%s Cannot be opened!",filename);
      return 1;
   }

   if (size>buffsize)
   {
      printf ("Not enough memory!");
      return 2;
   }

   f_read(buffer,size,1,file);
   f_close(file);

   return 0;
}
```

See also

f_wopen

2.34. f_findfirst

Find first file or subdirectory in specified directory. First call ***f_findfirst*** and if file was found get the next file with ***f_findnext***.

Format

```
int f_findfirst(const char *filename,F_FIND *find)
```

Arguments

Argument	Description
filename	name of file to find
find	where to store find information

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_INVALIDDIR	invalid path
F_ERR_NOTFOUND	file not found

Example

```
void mydir(void)
{
   F_FIND find;

   if (!f_findfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
         }
         else
         {
            printf (" size %d\n",find.len);
         }
      } while (!f_findnext(&find));
   }
}
```

See also

```
f_findnext
```

2.35. f_wfindfirst

Find first Unicode 16 file or subdirectory in specified directory. First call ***f_wfindfirst*** and if file was found get the next file with ***f_wfindnext***.

Format

```
int f_wfindfirst(const W_CHAR *filename, F_WFIND *find)
```

Arguments

Argument	Description
filename	Unicode 16 name of file to find
find	where to store find information

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_INVALIDDIR	invalid path
F_ERR_NOTFOUND	file not found

Example

```
void mydir(void)
{
   F_WFIND find;
   if (!f_wfindfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
         }
         else
         {
            printf (" size %d\n",find.len);
         }
      } while (!f_wfindnext(&find));
   }
}
```

See also

```
f_wfindnext
```

2.36. f_findnext

Find the next file or subdirectory in a specified directory after a previous call to ***f_findfirst*** or ***f_findnext***. First call ***f_findfirst;*** if file was found get the rest of the matching files by repeated calls to ***f_findnext***.

Format

```
int f_findnext(F_FIND *find)
```

Arguments

Argument	Description
find	find structure (from f_findfirst)

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file not found

Example

```
void mydir(void)
{
   F_FIND find;
   if (!f_findfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
         }
         else
         {
            printf (" size %d\n",find.len);
         }
      } while (!f_findnext(&find));
   }
}
```

See also

```
f_findfirst, f_findfirst
```

2.37. f_wfindnext

Find the next file or subdirectory in a specified directory after a previous call to ***f_wfindfirst*** or ***f_wfindnext***. First call ***f_wfindfirst***; if file was found get the rest of the matching files by repeated calls to ***f_wfindnext***.

Format

```
int f_wfindnext(F_WFIND *find)
```

Arguments

Argument	Description
find	find structure (from f_wfindfirst))

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file not found

Example

```
void mydir(void)
{
   F_WFIND find;
   if (!f_wfindfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
         }
         else
         {
            printf (" size %d\n",find.len);
         }
      } while (!f_wfindnext(&find));
   }
}
```

See also

```
f_wfindfirst, f_wfindfirst
```

2.38. f_settimedate

Set time and date on a file or on a directory.

A recommended format for the use of the time and date fields is given in the Real Time Clock section of Section 1.

> **Note:** The time/date data are simply two 16-bit numbers associated with the specified file that the developer is free to use as desired.

Format

```
int f_settimedate(const char *filename, unsigned short ctime, un-
      signed short cdate)
```

Arguments

Argument	Description
filename	file
ctime	creation time of file or directory
cdate	creation date of file or directory

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned short ctime,cdate;

   ctime = (15<<11)+(30<<5)+(23>>1);
   /* 15:30:22 */
   cdate = ((2002-1980)<<9)+(11<<5)+(3);
   /* 2002.11.03. */

   f_mkdir("subfolder");   /* creating directory */
   f_settimedate("subfolder",ctime,cdate);
}
```

See also

```
f_gettimedate
```

2.39. f_wsettimedate

Set time and date on a file or on a directory with Unicode 16 name.

A recommended format for the use of the time date fields is given in the Real Time Clock section of Section 1.

> **Note:** The time/date data is simply two 16-bit numbers associated with the specified file which the developer is free to use as desired.

Format

```
int  f_settimedate(const  W_CHAR  *filename,  unsigned  short  ctime,
      unsigned short cdate)
```

Arguments

Argument	Description
filename	Unicode 16 name of file
ctime	creation time of file or directory
cdate	creation date of file or directory

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned short ctime,cdate;

   ctime = (15<<11)+(30<<5)+(23>>1);
   /* 15:30:22 */
   cdate = ((2002-1980)<<9)+(11<<5)+(3);
   /* 2002.11.03. */

   f_wmkdir("subfolder");  /* creating directory */
   f_wsettimedate("subfolder",ctime,cdate);
}
```

See also

```
f_wgettimedate
```

2.40. f_gettimedate

Get time and date information from a file or directory. This field is automatically set by the system when a file or directory is created and when a file is closed.

> **Note**: The time/date data are simply two 16-bit numbers associated with the specified file. They can be used freely, as desired.

Format

```
int f_gettimedate(const char *filename, unsigned short *pctime,
      unsigned short *pcdate)
```

Arguments

Argument	Description
filename	target file or directory
pctime	pointer where to store the time
pcdate	pointer where to store the date

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory name contains invalid characters
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned short t,d;

   if (!f_gettimedate("subfolder",&t,&d))
   {
      unsigned short sec=(t & 0x001f) << 1;
      unsigned short minute=((t & 0x07e0) >> 5);
      unsigned short hour=((t & 0x0f800) >> 11);
      unsigned short day= (d & 0x001f);
      unsigned short month= ((d & 0x01e0) >> 5);
      unsigned short year=1980+((d & 0xfe00) >> 9);

      printf ("Time: %d:%d:%d",hour,minute,sec);
      printf ("Date: %d.%d.%d",year,month,day);
   }
   else
   {
      printf ("File time cannot retrieved!"
   }
}
```

See also

f_settimedate

2.41. f_wgettimedate

Get time and date information from a file or directory with a Unicode 16 name. This field is automatically set by the system when a file or directory is created and when a file is closed.

> **Note**: The time/date data are simply two 16-bit numbers associated with the specified file. They can be used freely, as desired.

```
int   f_wgettimedate(const   W_CHAR   *filename,   unsigned   short
      *pctime, unsigned short *pcdate)
```

Arguments

Argument	Description
filename	Unicode 16 name of target file or directory
pctime	pointer where to store the time
pcdate	pointer where to store the date

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory name contains invalid characters
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned short t,d;

   if (!f_wgettimedate("subfolder",&t,&d))
   {
      unsigned short sec=(t & 0x001f) << 1;
      unsigned short minute=((t & 0x07e0) >> 5);
      unsigned short hour=((t & 0x0f800) >> 11);
      unsigned short day= (d & 0x001f);
      unsigned short month= ((d & 0x01e0) >> 5);
      unsigned short year=1980+((d & 0xfe00) >> 9);

      wprintf ("Time: %d:%d:%d",hour,minute,sec);
      wprintf ("Date: %d.%d.%d",year,month,day);
   }
   else
   {
      wprintf ("File time cannot retrieved!"
   }
}
```

See also

f_wsettimedate

2.42. f_setpermission

This sets the file or directory permission field associated with a file.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the user and could, for instance, be used to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))
#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))
#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))
#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))
#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))
#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))
```

Format

```
int f_setpermission(const char *filename, unsigned long secure)
```

Arguments

Argument	Description
filename	target file
secure	32bit number to associate with filename

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   f_mkdir("subfolder");   /* creating directory */
   f_setpermission ("subfolder",0x00ff0000);
}
```

See also

f_getpermission

2.43. f_wsetpermission

This sets the file or directory permission field associated with a file with Unicode 16 name.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))
#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))
#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))
#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))
#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))
#define FSSEC_ATTR_READONLY (0x01UL<<(31-6)).
```

Format

```
int  f_wsetpermission(const  W_CHAR  *filename,  unsigned  long  se-
      cure)
```

Arguments

Argument	Description
filename	Unicode 16 name of target file
secure	32bit number to associate with filename

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory name contains invalid characters
F_ERR_INVALIDDRIVE	drive does not exist
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   f_mkdir("subfolder");   /* creating directory */
   f_wsetpermission ("subfolder",0x00ff0000);
}
```

See also

```
f_wgetpermission
```

2.44. f_getpermission

Retrieves file or directory permission field associated with a file.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))
#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))
#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))
#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))
#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))
#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))
```

Format

```
int f_getpermission(const char *filename, unsigned long *psecure)
```

Arguments

Argument	Description
filename	target file
psecure	pointer to where to store permission

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory contains invalid characters
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned long secure;

   if (!f_getpermission ("subfolder",&secure))
   {
      printf ("permission is: %d",secure);
   }
   else
      printf ("Permission cannot be retrieved!");
}
```

See also

```
f_setpermission
```

2.45. f_wgetpermission

Retrieves file or directory permission field associated with a file with a Unicode 16 name. Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system. The first six bits are reserved for use by the system, as follows:

```
#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))
#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))
#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))
#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))
#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))
#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))
```

Format

```
int   f_getpermission(const   W_CHAR   *filename,   unsigned   long
      *psecure)
```

Arguments

Argument	Description
filename	Unicode 16 name of target file
psecure	pointer to where to store permission

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_INVALIDNAME	file or directory contains invalid characters
F_ERR_NOTFOUND	file or directory not found

Example

```
void myfunc(void)
{
   unsigned long secure;
   if (!f_wgetpermission ("subfolder",&secure))
   {
      wprintf ("permission is: %d",secure);
   }
   else
      wprintf ("Permission cannot be retrieved!");
}
```

See also

f_wsetpermission

2.46. f_open

Open a file. The following open modes are allowed:

modes	description
"r"	Open an existing file for reading. The stream is positioned at the beginning of the file.
"r+"	Open an existing file for reading and writing. The stream is positioned at the beginning of the file/
"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file
"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file

Table 2, f_open modes

The same file can be opened multiple times in "r" mode.

If a file is opened in "w" or "w+" mode then there is a lock mechanism that denies opening the file in any other mode. This prevents opening the file for reading and writing at the same time.

If a file is open in "a" or "a+" mode, then any number of "r" mode opens are allowed at the same time.

> **Note:** There is no text mode. The system assumes all files to be accessed in binary mode only.

Format

```
F_FILE *f_open(const char *filename, const char *mode);
```

Arguments

Argument	Description
filename	target file
mode	open mode

Return values

Return value	Description
F_FILE *	pointer to the associated opened file or zero if it could not be opened

Example

```
void myfunc(void)
{
   F_FILE *file;
   char c;

   file=f_open("myfile.bin","r");
   if (!file)
   {
      printf ("File cannot be opened!");
      return;
   }

   f_read(&c,1,1,file); /* read 1byte */
   printf ("'%c' is read from file",c);
   f_close(file);
}
```

See also

f_read, f_write, f_close,

2.47. f_wopen

Open a file with Unicode 16 filename. The following open modes are allowed:

modes	description
“r”	Open an existing file for reading. The stream is positioned at the beginning of the file.
“r+”	Open an existing file for reading and writing. The stream is positioned at the beginning of the file.
“w”	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
“w+”	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
“a”	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
“a+”	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file

Table 3, fw_open modes

The same file can be opened multiple times in “r” mode.

If a file is opened in “w” or “w+” mode then there is a lock mechanism that denies opening the file in any other mode. This prevents opening the file for reading and writing at the same time.

If a file is open in “a” or “a+” mode, then any number of “r” mode opens are allowed at the same time.

> **Note:** There is no text mode. The system assumes all files to be accessed in binary mode only.

Format

```
F_FILE *f_wopen(const W_CHAR *filename, const char *mode);
```

Arguments

Argument	Description
filename	Unicode 16 name of target file
mode	open mode

Return values

Return value	Description
F_FILE *	pointer to the associated opened file or zero if it could not be opened

Example

```
void myfunc(void)
{
   F_FILE *file;
   char c;

   file=f_wopen("myfile.bin","r");
   if (!file)
   {
      wprintf ("File cannot be opened!");
      return;
   }

   f_read(&c,1,1,file); /* read 1byte */
   wprintf ("'%c' is read from file",c);

   f_close(file);
}
```

See also

f_read, f_write, f_close

2.48. f_truncate

Opens a file for writing and truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

Format

```
F_FILE *f_truncate(const char *filename, unsigned long length);
```

Arguments

Argument	Description
filename	file to be opened
length	new length of file

Return values

Return value	Description
F_FILE *	pointer to the associated opened file handle or zero if it could not be opened

Example

```
int mytruncatefunc(char *filename, unsigned long length)
{
   F_FILE *file=f_truncate(filename,length);

   if(!file)
      printf("File not found");
   else
   {
      printf("File %s truncated to %d bytes,
      filename, length);
      f_close(file);
   }
   return 0;
}
```

See also

```
f_open
```

2.49. f_ftruncate

If a file is opened for writing, then this function truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

Format

```
int f_ftruncate(F_FILE *filehandle, unsigned long length);
```

Arguments

Argument	Description
filehandle	open file handle
length	new length of file

Return values

Return value	Description
F_NO_ERROR	success
else	see error codes

Example

```
int mytruncatefunc(F_FILE *file, unsigned long length)
{
   int ret=f_ftruncate(filename,length);

   if (ret)
   {
      printf("error:%d\n",ret);
   }
   else
   {
      printf("File is truncated to %d bytes", length);
   }

   return ret;
}
```

See also

```
f_open, f_truncate
```

2.50. f_wtruncate

Opens a file for writing and truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

Format

```
F_FILE   *f_wtruncate(const   W_CHAR   *filename,   unsigned   long
      length);
```

Arguments

Argument	Description
filename	file to be opened
length	new length of file

Return values

Return value	Description
F_FILE *	pointer to the associated opened file handle or zero if it could not be opened

Example

```
int mywtruncatefunc(W_CHAR *filename, unsigned long length)
{
   F_FILE *file=f_wtruncate(filename,length);
   if(!file)
      printf("File not found");
   else
   {
      printf("File %s truncated to %d bytes,
      filename, length);
      f_close(file);
   }

   return 0;
}
```

See also

f_wopen

2.51. f_close

Close a previously opened file.

Format

```
int f_close(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_NOTOPEN	file not open
F_ERR_INVALIDDRIVE	file handle points to invalid drive
F_ERR_ONDRIVE	cannot write into device

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string="ABC";

   file=f_open("myfile.bin","w");

   if (!file)
   {
      printf ("File cannot be opened!");
      return;
   }

   f_write(string,3,1,file); /* write 3byte */
   if (!f_close(file))
   {
      printf ("File stored");
   }
   else printf ("file close error");
}
```

See also

```
f_open, f_read, f_write
```

2.52. f_flush

Flush data to the media. This command allows the user to update the file on the media and therefore update the failsafe state of the file without again closing and opening the file. Once this command has completed, the new state of the file will be restored after system restarting or a system failure.

Format

```
int f_flush(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_NOTOPEN	file not open
F_ERR_INVALIDDRIVE	file handle points to invalid drive
F_ERR_ONDRIVE	cannot write into device

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string="ABC";

   file=f_open("myfile.bin","w");
   if (!file)
   {
      printf ("File cannot be opened!");
      return;
   }

   f_write(string,3,1,file); /* write 3byte */

   if (!f_flush(file))
   {
      printf ("New data is now failsafe");
   }
   else printf ("file flush error");
}
```

See also

```
f_open, f_write, f_close
```

2.53. f_write

Write data into file at current position. File must be opened with "r+", “w”, “w+”, "a+" or “a”. The file pointer is moved forward by the number of bytes successfully written.

> **Note:** Data is NOT permanently stored to the media until either an ***f_flush*** or ***f_close*** has been executed on the file.

Format

```
long f_write(const void *buf, long size, long size_st, F_FILE
      *filehandle)
```

Arguments

Argument	Description
buf	buffer where data is
size	size of items to be written
size_st	number of items to be written
filehandle	file handle to write to

Return values

Return value	Description
number	number of items written

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string=”ABC”;

   file=f_open(“myfile.bin”,”w”);
   if (!file)
   {
      printf (“File cannot be opened!”);
      return;
   }
   if (f_write(string,1,3,file)!=3)
   {  /* write 3bytes */
      printf (“not all items written”);
   }
   f_close(file);
}
```

See also

```
f_read, f_open, f_close, f_flush
```

2.54. f_read

Read data from the current file position. File must be opened with “r”, "r+", "w+" or "a+". The file pointer is moved forward by the number of bytes read.

Format

```
long   f_read(void   *buf,   long   size,   long   size_st,   F_FILE
      *filehandle)
```

Arguments

Argument	Description
buf	buffer where to store data
size	size of each of items to be read
size_st	number of items to be read
filehandle	file handle to read it

Return values

Return value	Description
number	number of items read

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,”r”);

   long size=f_filelength(filename);
   if (!file)
   {
      printf (“%s Cannot be opened!”,filename);
      return 1;
   }
   if (f_read(buffer,1,size,file)!=size)
   {
      printf (“not all items read”);
   }
   f_close(file);
   return 0;
}
```

See also

```
f_seek, f_tell, f_open, f_close, f_write
```

2.55. f_seek

Move read/write position in the file. `Whence` parameter could be one of:

```
F_SEEK_CUR - Current position of file pointer
F_SEEK_END - End of file
F_SEEK_SET - Beginning of file
```

Offset position is relative to whence. If seek position is beyond the end of file, then data will be padded with zeroes.

Format

```
int f_seek(F_FILE *filehandle,long offset, long whence)
```

Arguments

Argument	Description
filehandle	handle of target file
offset	relative byte position according to whence
whence	where to calculate offset from

Return values

Return value	Description
F_NO_ERROR	success
F_ERR_NOTFORREAD	file not open for reading
F_ERR_NOTUSEABLE	whence parameter is invalid
F_ERR_ONDRIVE	drive is not readable
F_ERR_INVALIDDRIVE	invalid drive specified in file handle

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");
   f_read(buffer,1,1,file); /* read 1 byte */
   f_seek(file,0,F_SEEK_SET);
   f_read(buffer,1,1,file);/*read the same 1 byte */
   f_seek(file,-1,F_SEEK_END);
   f_read(buffer,1,1,file); /* read last 1 byte */
   f_close(file);
   return 0;
}
```

See also

`f_read, f_tell`

2.56. f_tell

Tell the current file position in the target file.

Format

```
long f_tell(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
filepos	current read or write file position

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");

   printf ("Current position %d",f_tell(file));
   f_read(buffer,1,1,file); /* read 1 byte */
   printf ("Current position %d",f_tell(file));
   f_read(buffer,1,1,file); /* read 1 byte */
   printf ("Current position %d",f_tell(file));

   f_close(file);
   return 0;
}
```

See also

f_seek, f_read, f_write, f_open

2.57. f_seteof

Move the end of file to the current file position. All data after the new EOF position are lost.

Format

```
int f_seteof(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	handle of open target file

Return values

Return value	Description
0	Success
else	see error codes

Example

```
int mytruncatefunc(char *filename, int position)
{
   F_FILE *file=f_open(filename,"r");

   f_seek(file,position,F_SEEK_SET);

   if(f_seteof(file))
   {
      printf("Truncate Failed\n");
      return 1;
   }

   f_close(file);
   return 0;
}
```

See also

```
f_truncate, f_write, f_open
```

2.58. f_eof

Check whether the current position in the opened target file is the end of the file.

Format

```
int f_eof(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
0	not at end of file
else	end of file or invalid file handle

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");

   while (!f_eof())
   {
      if (!buffsize) break;
      buffsize--;
      f_read(buffer++,1,1,file);
   }

   f_close(file);
   return 0;
}
```

See also

f_seek, f_read, f_write, f_open

2.59. f_rewind

Set the current file position in the open target file to the beginning.

Format

```
int f_rewind(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
0	success
else	failed: invalid file handle

Example

```
void myfunc(void)
{
   char buffer[4];
   char buffer2[4];

   F_FILE *file=f_open("myfile.bin",”r”);

   if (file)
   {
      f_read(buffer,4,1,file);
      f_rewind(file); /* rewind file pointer */
      f_read(buffer2,4,1,file);
      /* read from beginning */
      f_close(file);
   }
   return 0;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

2.60. f_putc

Write a character to the open target file at the current file position. The current file position is incremented.

Format

```
int f_putc(int ch,F_FILE *filehandle)
```

Arguments

Argument	Description
ch	character to be written
filehandle	file handle of target

Return values

Return value	Description
-1	write Failed
Value	successfully written character

Example

```
void myfunc (char *filename, long num)
{
   F_FILE *file=f_open(filename,"w");

   while (num--)
   {
      int ch='A';
      if(ch!=(f_putc(ch))
      {
         printf("f_putc error!");
         break;
      }
   }
   f_close(file);
   return 0;
}
```

See also

```
f_seek, f_read, f_write, f_open
```

2.61. f_getc

Read a character from the current position in the open target file.

Format

```
int f_getc(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
value	character read from file or -1 if error

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");
   while (buffsize--)
   {
      int ch;
      if((ch=f_getc(file))== -1)
         break;
      *buffer++=(char)ch;
      buffsize--;
   }

   f_close(file);
   return 0;
}
```

See also

```
f_seek, f_read, f_write, f_open, f_eof
```

2.62. f_stat

Get information about a file. This function retrieves information by filling the F_STAT structure passed to it. It sets filesize, creation time/date, the drive number in which the file is located, and secure attributes.

Format

```
int f_stat (const char *filename, F_STAT *stat);
```

Arguments

Argument	Description
filename	file
stat	pointer to F_STAT structure to be filled

Return values

Return value	Description
F_NO_ERROR	success
else	see error codes

Example

```
void myfunc(void)
{
   F_STAT stat;
   if (f_stat("myfile.txt",&stat))
   {
      printf ("error");
      return;
   }
   printf ("filesize:%d",stat.filesize);
}
```

See also

```
f_gettimedate, f_settimedate
```

2.63. f_checkvolume

This function is used to check the status of a drive that has been initialized.

Format

```
int f_checkvolume(int drivenum)
```

Arguments

Argument	Description
drivenum	drive to be checked (0:A, 1:B...)

Return values

Return value	Description
F_NO_ERROR	drive is working
else	there is an error on the drive; e.g., card is missing

Example

```
void mychkfs(int num)
{
   int ret;

   /*Delete volume 1 */
   if(f_checkvolume(num))
   {
      printf("Volume %d is not usable, Error %d", num, ret);
   }
   else
   {
      printf(("Volume %d is working", num);
   }
      .
      .
}
```

See also

```
f_mountdrve, f_unmountdrive
```

2.64. f_get_oem

This returns the OEM name “CMX_SAFE_FS”. The pointer passed for storage should be capable of holding a 12-character string.

Format

```
int f_get_oem(int drivenum, char *str, long len)
```

Arguments

Argument	Description
drivenum	drive number
str	pointer to copy label to
len	length of storage area

Return values

Return value	Description
F_NOERROR	success
else	see error codes

Example

```
void get_disk_oem(void)
{
   char oem_name[12];
   int result;

   oem_name[11]=0;   /* zero terminate string */
   result = f_get_oem(f_getcurrdrive(),oem_name,12);

   if (result)
      printf("Error on Drive");
   else
      printf("Drive OEM is %s",oem_name);
}
```

3. NOR Flash Driver

3.1. Physical Device Usage

The developer must make some decisions about how to use the flash device, and must be aware that all flash devices are divided into a set of erasable blocks. It is possible only to write to an erased location; it is not possible to erase anything smaller than a block. Therefore, some complex management software is used. On some devices the size of the erasable blocks may vary.

> **Note**: The fsmem.exe utility should be used to help you to understand the usage of the blocks and to make it easier to derive the optimum solution for your requirements.

SAFE operates on a set of logical blocks that may be further divided into sectors. The physical driver must do two things in this respect:

1. It defines for the file system which logical block numbers are to be used for what purpose; this is configured in the FS_FLASH structure and returned to the file system by the ***fs_phy_nor_xxx*** function.

2. It provides a mapping between the logical block numbers used by the file system and the physical addresses of the blocks in the flash device (this is done by the ***GetBlockAddr*** function).

The user has three types of blocks to assign to the device:

- Reserved blocks - for processes other than the file system; e.g., booting
- Descriptor blocks - to hold information about the structure of the file system, wear, etc. By using a minimum of 2 descriptor blocks (and management software) the system is failsafe.
- File system blocks - for storing file information.

The sections below describe how to assign these and provide worked examples.

3.1.1. Reserved blocks

Blocks can be reserved for private usage without restriction. This is done simply by omitting those blocks from the ***GetBlockAddr*** function.

Reserved blocks may be accessed by using the ***GetBlockAddr*** function and also by selecting the physical block numbers to be used and ensuring that they are not specified in the descriptor and file system usage described below.

> **Note:** Care should be taken in accessing reserved blocks and attention should be paid to the specification of the device used to ensure interoperability. Some devices allow an erase operation to be performed while another block is being read; others have different rules. In general it is sensible to use only the file system or the reserved sectors at any one time. In any case, careful understanding of the specific device is required.

3.1.2. Descriptor Blocks

(see also the "Sectors and File Storage" section below)

These blocks contain critical information about the file system, including block allocation, wear information, caching information and file/directory information. At least two descriptor blocks that can be erased independently must be included in the system. An optional descriptor writing cache may be configured; this improves the performance of the file system.

On a flash device with different sized blocks it is generally sensible to use some of the smaller blocks as descriptor blocks. This also improves the performance of the system. However, when using the cache this is not so important and it is preferable to allocate a larger cache.

The definitions for parameters that must be set up in the NOR physical header file are listed below:

DESCSIZE

This is the size of a descriptor block. All descriptor blocks must be the same size. There may be only one descriptor in a single physical block. A descriptor must be large enough to store the specified write cache (see DESCCACHE below) plus all the information about directory entries and files, as well as block and sector information. A calculator program **(/util/fsmem.exe**) is provided with the package to help you work out the effect of setting a particular descriptor size.

> **Note:** where RAM usage is a consideration it is also possible to set the descriptor size to less than the physical block size, so long as it fits in a single physical block that is used only for this purpose.

DESCBLOCKSTART

This is the logical number of the first block to be used by the file system as a descriptor block.

DESCBLOCKS

This is the number of descriptor blocks to be used by the file system. At least two descriptor blocks must be defined.

DESCCACHE

This defines the descriptor write cache size. This number must be less than DESCSIZE, since the cache is allocated in the descriptor block. If it is set to zero the descriptor-write-cache method will not be used. The descriptor write cache is an efficient method of up-

dating the changes in the descriptor, since the whole descriptor need not be re-written, while the 100% power-fail safe characteristics of the system will still be retained.

Use of the descriptor write cache reduces to an absolute minimum the wear-leveling and the number of erases required when updating the system.

It is highly recommended to use the descriptor write cache since performance and wear characteristics of the system are improved by a larger cache. However, a larger cache size also reduces the number of directory entries; use the **fsmem.exe** utility to check the effect of this.

3.1.3. File System Blocks

The developer should allocate as many of these as required for file storage.

The parameters that must be set up in ***fs_phy_nor_xxx*** are listed below:

MAXBLOCKS

This defines the number of erasable blocks that are available for file storage.

BLOCKSTART

This defines the logical number of the first of the block that may be used for file storage. This is the logical number used when the ***GetBlockAddr*** function is called.

BLOCKSIZE

This defines the size of the blocks to be used in the file storage area. It be an erasable unit of the flash chip. All blocks in the file storage area must be the same size. This may be different from the DESCSIZE (see above) where the flash chip has erasable units of different sizes.

SECTORSIZE

This defines the sector size. Each block is divided into a number of sectors. **SECTORSIZE** is the smallest usable unit in the system and thus represents the minimum file storage area. For best usage of the flash blocks the sector size should always be a power of 2. For more information see the sector section below.

SECTORPERBLOCK

This defines the number of sectors in a block. It must always be true that:

```
SECTORPERBLOCK = BLOCKSIZE / SECTORSIZE
```

3.1.4. Example 1

The target flash device (e.g., AM29LV160B – see the **29lv160b.c** file for reference) has 35 erasable blocks (1x16K, 2x8K, 1x32K, 31x64K) and the user wants to reserve blocks 0 and 3 for private usage. A possible configuration is:

```
BLOCKSIZE       64K   size of file storage blocks
BLOCKSTART      4     logical first file storage block (4-18 used)
MAXBLOCKS       31    number of blocks for use by file storage

DESCSIZE        8K    descriptor size
DESCBLOCKSTART  1     logical first descriptor block number
DESCBLOCKS      2     number of descriptor blocks
DESCCACHE       2K    set a write cache of 2K
```

The table below shows how the physical/logical blocks are arranged:

Physical Block Number	Physical Block Size	Logical Block Number	Usage
0	16k	0	Reserved block
1	8k	1	Descriptor block
2	8k	2	Descriptor block
3	32k	3	Reserved block
4…34	64K	4-34	File storage blocks

Table 4, Example 1 - arrangement of physical/logical blocks

Thus ***GetBlockAddr*** algorithm for this could be:

```
unsigned long GetBlockAdd (long block, long relsector)
{
   if(block==0)      /* free/unused block */
      return(0);
   if(block==1)      /* descriptor block */
      return(16K);
   if(block==2)      /* descriptor block */
      return(16K+8K);
   if(block==3)      /* free/unused block */
      return(16K+8K+8K);

   /* file system blocks */
   return(16K+8K+8K+32K+(block-BLOCKSTART)*BLOCKSIZE)+
         (relsector*SECTORSIZE));
}
```

3.1.5. Example 2

Using a flash device with 512*128K erasable blocks (e.g., AM29LV2562M – see the **29lv2562m.c** file for reference). A minimum of two erasable blocks must be used for descriptors. These blocks are quite large. Therefore it is a good idea to define a large part of this for a write cache; in this example we will create a 32K cache. Using a cache of this size cache has two advantages: the number of required erases is reduced, and the wear on the device is reduced.

We then decide to use the remaining 510 physical blocks for file system storage. So a configuration could look like:

```
BLOCKSIZE      128K  size of file storage blocks
MAXBLOCKS      510   number of blocks for use by file storage
BLOCKSTART     0     logical first file storage block (0 – 509
                     are used)
DESCSIZE       128K  descriptor size (4 per physical block)
DESCBLOCKSTART 510   logical first descriptor block number
DESCBLOCKS     2     number of descriptor blocks
DESCCACHE      32K   size of write descriptor cache
```

The table below shows how the physical/logical blocks are arranged:

Physical Block Number	Physical Block Size	Logical Block Number	Usage
0-509	64k	0-509	File storage blocks
510-511	64k	510-511	Descriptors

Table 5, Example 2 - arrangement of physical/logical blocks

The code below shows possible modifications to the driver:

Thus ***GetBlockAddr*** algorithm for the above could be:

```
unsigned long GetBlockAdd (long block, long relsector)
{
   return((block*BLOCKSIZE)+(relsector*SECTORSIZE));
}
```

3.2. Sectors and File Storage

The blocks of the file storage section of the file system are sub-divided into sectors of equal size. These sectors are the minimum writeable area on the device and are the minimum area taken up by a file. For file systems with many small files it is advantageous to keep the sector size small to maximize the number of files that may be stored in the system. An additional benefit is that if the files are small, then many more can be written before a block erase is required.

For example, if there is 1 sector per block, then a block must be erased for every file. However, if there are 32 sectors per block then 32 small files can be written before it is necessary to erase another block.

There is, however, a balance to be struck between the maximum number of files and the number of sectors in the system.

> **Note**: The **/utils/fsmem.exe** utility should be used to help you to understand the usage of the blocks and to make it easier to derive the optimum solution for your requirements.

A descriptor block must contain:

> Block descriptors (6 bytes each)
> Sector descriptors (2 bytes each)
> File descriptors (32 bytes each)

Thus the maximum number of files allowed in the system may be given by the formula

```
Max Files < ((DescSize-DescCache) - 6*Maxblock –
               2*Maxblock*sectorperblock)/32
```

The developer should find a balance between having many sectors per block and allowing enough space in the descriptor for the required number of file descriptors.

If a balance cannot be found larger descriptor blocks should be considered, but this comes with a penalty: the erase time of the frequently-used descriptor blocks will increase.

Again, the **fsmem.exe** should be used for calculating the capabilities of a particular file system on the basis of input configuration information.

> **Note:** If files with long names are used, the total number of files that can be stored will be reduced.

3.3. Files

The NOR flash interface to the file system requires two files:

Flashdrv.c - device-independent flash control layer
29lvxxx.c - physical chip controller

The **flashdrv.c** module provides a single clean interface between the intermediate file system and the physical chip. This module gets information about the configuration of the underlying flash chip, and the interface routines to call from the **29lvxxx.c** module. Then it builds a controller based on that information. This module also does the wear-level control for the device.

> **Note**: Normally this module does not require modification. If modification is required it is strongly recommended that CMX Systems should be contacted about any particular or unusual requirements.

The **29lvxxx.c** module depends on the specific flash device and its configuration. Relevant data are the manufacturer, chip size, number of interface bits (8, 16 or 32), number of chips in parallel. All of these factors influence the code in this module.

The ***fs_phy_nor_29lvxxx*** function is the key to understanding the interface between the specific physical driver and the file system. The structure returned by this call contains all configuration information about block usage required by the upper layers as well as the set of interface function pointers to be used. The module provides the following interface functions to the **flashdrv.c** module through the FS_FLASH structure:

NOR flash functions:

- ReadFlash
- EraseFlash
- WriteFlash
- VerifyFlash (optional)
- BlockCopy (required only if static wear is used)

The only public function in this module is ***fs_phy_nor_29lvxxx***, which must be passed to the ***f_mountdrive*** API function to initialize the physical driver.

These functions require subroutine calls to do their work. After the function definitions a description of all the routines used in this module is given. These routines are documented for an AMD 29LV320B NOR flash chip. Implementation may vary for other devices. The routines are documented to give guidance as to how to implement this module.

3.4. Physical Interface Functions

The functions in this section provide the interface to the upper layer and must be ported to meet the requirements of the particular flash devices that are used. A sample driver for an AMD29Lxx device is supplied for reference purposes.

3.4.1. fs_phy_nor_xxx

This is the first call made by the upper layer to discover the flash device configuration. This function can be used for initializing the flash device, and also for detecting the flash type. It gives information to the upper layer about the number of blocks, block sizes, sector size, cache size, etc.

Format

```
int fs_phy_nor_xxx(FS_FLASH *flash)
```

Arguments

Argument	Description
flash	flash structure that is needed to be filled

Return values

Return value	Description
0	if flash device successfully checked
any number	if there was any error during initialization

Comments

This is the FS_FLASH structure that the module configures:

```
typedef struct {
   long maxblock; /*maximum number of block can be used */
   long blocksize;   /*block size in bytes */
   long sectorsize;  /*sector size to use */
   long sectorperblock; /*    sector/block    (block    size/sector
size)*/
   long blockstart;   /* where  physical block start */
   long descsize;     /* max size of fat+directory+block index */
   long descblockstart; /* where to store 1st descriptor block */
   long descblockend;  /* where to store last descriptor block */
   long separatedir;    /* not used for NOR */
   long cacheddescsize;    /* size of descriptor write cache */
   long cachedpagenum;     /* not used in NOR */
   long cachedescpagesize; /* not used in NOR */
   FS_PHYREAD   ReadFlash;    /* read content fn pointer */
   FS_PHYERASE  EraseFlash;   /* erase a block fn pointer */
   FS_PHYWRITE  WriteFlash;   /* write content fn pointer */
   FS_PHYVERIFY VerifyFlash;  /* verify content fn pointer */
   FS_PHYCHECK  CheckBadBlock;      /* not used for NOR */
   FS_PHYSIGN   GetBlockSignature;  /* not used for NOR */
   FS_PHYCACHE WriteVerifyPage;     /* not used in NOR */
   FS_PHYBLKCPY BlockCopy;    /* HW/SW accelerated block copy */
   unsigned char *chkeraseblk;  /* buffer for preerasing blocks
                                   optional request to erase */
   unsigned char *erasedblk;   /* buffer for preerasing blocks
                               optional set to erased */
} FS_FLASH;
```

3.4.2. ReadFlash

This function is called from the higher layer to read data from flash..

Format

```
int ReadFlash(void *data, long block, long blockrel, long
      datalen)
```

Arguments

Argument	Description
data	pointer data storage area
block	zero based block number to be read
blockrel	relative position in block where to start reading
datalen	length of data to be read

Return values

Return value	Description
0	success
else	error during read

Comments:

`Blockrel` is a number that provides the position at which reading starts; it can range from 0 to the block size.

`Datalen` is always less than block size and never points out from a given block, even if blockrel points into the middle of the block.

3.4.3. EraseFlash

Erase a block in flash.

Format

```
int EraseFlash(long block)
```

Arguments

Argument	Description
block	zero based block number to be erased

Return values

Return value	Description
0	successfully erased
any number	error during erasing

3.4.4. WriteFlash

Write data into the flash device.

Format

```
int WriteFlash(void *data, long block, long relsector, long size,
      long relpos)
```

Arguments

Argument	Description
data	points to source data to be written
block	zero based block number where to store data
sector	zero based relative sector number in block
size	length of data to be stored
relpos	relative position in block to write data

Return values

Return value	Description
0	successfully written
any number	if there was any error during writing

3.4.5. VerifyFlash

This function is called from a higher level to verify data written after ***WriteFlash***. The incoming parameters are the same as for ***WriteFlash***. This function is for comparing written data with the original.

Format

```
int VerifyFlash(void *data, long block, long relsector, long
      size, long relpos)
```

Arguments

Argument	Description
data	points source data to be compared
block	zero-based block number of data to be compared
relsector	zero-based relative sector number in block
size	length of data to be compared
relpos	relative position in block of data to verify

Return values

Return value	Description
0	successfully verified
any number	if there was any error during verifying

Comment:

The verify function is not always necessary; it depends on the particular flash chip and what is specified in the datasheet to guarantee that a program operation has completed successfully. If this function is not needed, then it should return with 0.

3.4.6. BlockCopy

This function copies one block to another block. It is called only if static wear leveling is being used. This routine should be implemented to use any features of the target device that may be available to accelerate a block-to-block copy operation. Many devices have features to support block copy. They help to reduce CPU load and improve system performance. See Static Wear section for further details.

Format

```
int BlockCopy(long destblock, long soublock)
```

Arguments

Argument	Description
destblock	block number to copy to
soublock	block number to copy from

Return values

Return value	Description
0	success
else	failed

3.5. Subroutine Descriptions and Notes for Sample Driver

This section contains a complete list of subroutines, with a description of their functionalities. It includes notes for porting the routines to a particular hardware design.

FS_FLASHBASE

This define specifies the base address for accessing the Flash memory array. The value can be determined only from the hardware design. Sample code is based on an ARM implementation and reads the value from the Flash chip select.

RemoveWriteProtect

Remove hardware-supported write protect from Flash's chip select. The developer may supply another function that is based on the particular hardware design. If write protection is not required this function may be left empty.

SetWriteProtect

Set hardware-supported write protection to Flash's chip select (prevention for further writing). The developer may supply another function that is based on the particular hardware design. If write protection is not required this function may be left empty.

GetBlockAddr(block: long, relsector: long)

Calculate physical address of relative sector in specified block. When a descriptor block is specified, the sector field should be ignored and the base address of the block should be returned.

This routine must be modified to return the correct block/sector addresses for the requested logical blocks that have been set up in the ***fs_phy_nor_vxxx*** routine.

WriteCmd(cmd: ushort)

Write command sequence to flash device (0x555, 0xaa; 0x2aa, 0x55; 0x555, cmd). This command must be modified so that it's appropriate for the specific type of flash device being used. The sample program is for an AM29xxxx series flash device.

DataPoll(addr: long, chk ushort)

This is an AMD-specific sub-routine for checking that data have been written correctly. The algorithm is:

```
for
   if timeout reached return 2   /* Timeout error */
   readdata from flash addr
   if (data == chk) return 0     /* Ok */
   if (no poll needed) check data and return ok or data error
end for
```

***EraseFlash*(block: long)**

This routine is used by the higher level software to erase a logical block of flash memory.

The basic algorithm is:

```
addr = GetBlockAddr(block, 0)
RemoveWriteProtect()
Send Erase Command and addr of which block need to be erased
SetWriteProtect()
return DataPoll(addr)  /* wait until erase is finished and return
with result */
```

The commands must be modified so they are appropriate for the specific type of flash device being used. The sample program is for an AM29xxxx series flash.

WriteFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

This routine is called by the higher levels to write data to the flash device. Note: The **sdata** parameter is not used.

Algorithm:

```
Destaddr = GetBlockAddr(block, relsector)
Do 16bit data length align
RemoveWriteProtect()
for
   Send Write Command to flash device and program 16bit
   If (DataPoll(addr,data)) return error
   /* wait program end, if error returns */
   If length is reached then end of programming
end for
exit program mode by sending exit command to flash device
SetWriteProtect()
Return ok
```

The commands must be modified so that they are appropriate for the specific type of flash device being used. The sample program is for an AM29xxxx series flash device.

VerifyFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

This routine is called by the higher levels after a write operation has been completed to ensure that the data has been written correctly. Note: the **sdata** parameter is not used.

Algorithm:

```
Addr = GetBlockAddr(block, relsector) + Flash base
Do 16bit data length align
Verify programmed data with original data, if error then returns
with error
If  all data is checked returns with no error
```

The commands must be modified so that they are appropriate for the specific type of flash device being used. The sample program is for an AM29xxxx series flash device.

ReadFlash(data: ptr, block: long, blockrel: long, datalen: long)

This routine reads the specified amount of data from the flash device.

Algorithm:

```
Addr = GetBlockAddr(block, 0 ) + Flash base
Calculating start position from blockrel
Copy all data onto data address from flash device
```

The commands must be modified so that they are appropriate for the specific type of flash device being used. The sample program is for an AM29xxxx series flash device.

fs_phy_nor_29lvxxx (flash: struct)

Initialises internal functions to the flash structure.

Algorithm:

```
RemoveWriteProtect()
Get device ID and manufacturer ID from the flash
SetWriteProtect()
Compare read values with all supported device/manufacture codes
and fill the flash structure with corresponding data (size, sec-
tors, block information)
If a matching device is not found return with error
```

fnWriteWord (base: ptr, addr: long, data: ushort)

Add the flash relative address to the base pointer, and write 16 bits of data into the flash. This function is in the 29lxxxx.s file and is written for an ARM-based system with 16-bit access to the flash. This and calls to it must be modified according to the hardware design.

3.6. Pre-Erase and Erase Suspend/Resume

The driver can be designed to pre-erase unused blocks of flash. The erase operations can be resumed and suspended by read and write operations that will reduce system latency.

Pre-erasing dirty blocks will improve the performance of your system greatly.

Pre-erase can be done only on devices that have commands for suspending and resuming the erase operation. Since erase operations can take several seconds on NOR, for some NOR flash devices this option can greatly reduce system latencies. The host system must have some form of task switching and a priority mechanism to support this feature.

> **Note**: The sample driver for Intel StrataFlash (28f128j3pre.c) has sample code that implements this logic. The following describes how to implement this feature based on this driver sample.

File system calls will operate at higher priority than the LowFlashErase task; otherwise the file system operation would be forced to wait on all blocks that can be pre-erased to be erased. By running at a higher priority, read or write on the flash will cause suspension of erase operations, the read or write operation will be completed, and then the erase operation will be resumed.

In the sample driver the function ***LowFlashErase()*** should be called regularly by a task with priority lower than applications accessing the file system. This function searches for a block that may be pre-erased and will erase any it finds. It will return when no more erasable blocks are found. Only the flash access parts of this function should be changed.

To control access to the flash device a mutex must be created to ensure controlled access – this is ***gl_mutex*** in the sample driver. The mutex logic should not be changed.

Two functions must also be added to the driver:

1. ***SuspendErase()*** – sends a command to the flash to suspend the erase operation
2. ***ResumeErase()*** – sends a command to the flash to resume a suspended erase operation

Both the ***WriteFlash()*** and ***ReadFlash()*** functions must be modified to get the ***gl_mutex*** and call ***SuspendErase()*** before writing or reading. When the read or write is complete, the write and read functions call ***ResumeErase()*** and put the ***gl_mutex***.

In the **flash init** function, FS_FLASH structures **chkeraseblk** and **erasedblk** must be filled with buffers in which the status data of the pre-erased blocks are stored.

4. Atmel DataFlash™ Driver

These devices have their own particular characteristics; therefore, for reliable operation, drivers must be designed specifically for them.

The main features are:

- Data are written to a page buffer in the RAM of the device. Before it is written the underlying buffer is erased. This means that in the event of power-loss the whole or part of a page can be lost.

- If there have been 10,000 write operations to a particular sector, then the system must ensure that every page in that sector has been written to during that time; otherwise data loss may occur.

This driver handles all DataFlash issues to provide an efficient and failsafe interface for using these devices.

The interface is straightforward. Include the **dfdrv.c**, **s_atmel.c** and **spi.c** files and their headers in your project and follow the sections below.

4.1. DataFlash Configuration

> **Note**: The **fsmem.exe** utility should be used to help you to understand the usage of these settings and to make it easier to derive the optimum solution for your requirements.

First define your device type in **s_atmel.h** from those listed:

```
AT45DB11B
AT45DB21B
AT45DB41B
AT45DB81B
AT45DB161B
AT45DB321B
AT45DB642B
```

If your device is not one of those listed, contact support@cmx.com.

The device type definition will automatically set the following parameters to their default values for the specified chip:

```
ADF_PAGE_SIZE
ADF_REAL_PAGE_COUNT
ADF_NUM_OF_SECTORS
ADF_PAGES_PER_SECTOR
ADF_BYTE_ADDRESS_WIDTH
F_ATMEL_DEFAULT_SECTORS_PER_BLOCK
F_ATMEL_DEFAULT_CACHED_DESC
F_ATMEL_DEFAULT_NO_OF_DESC_BLOCKS
```

The default values comprise a tested configuration that uses the entire target device. These settings should not be changed without very good reason.

From these settings are derived some standard definitions for the driver:

F_ATMEL_SECTORSIZE

The sector size to be used by the file system. It should always be a multiple of 8 pages.

F_ATMEL_BLOCKSIZE

The block size to be used by the file system. It should always be the sector size multiplied by a power of 2.

F_ATMEL_CACHED_DESC

Number of cache descriptors.

F_ATMEL_NO_OF_DESC_BLOCKS

Number of descriptor blocks.

4.2. Physical Interface Functions

The physical interface functions:

s_atmel_flash_fs_phy()
adf_read_flash()
adf_erase_flash()
adf_write_flash()
adf_verify_flash()
adf_block_copy()

work identically to the corresponding functions in the NOR flash driver. Please see Physical Interface Functions for detailed information.

4.3. Porting the SPI interface

The physical interface to the DataFlash is through SPI. A sample SPI driver is provided. Because all platforms implement their SPI interfaces in different ways, the driver must be ported to work with your target platform.

To use the sample driver, include the **spi.c** and **spi.h** files in your project. The higher level uses just 5 interface functions to the SPI. The ported sample driver must provide a logically identical interface.

In **spi.h** the following functionality must be replicated:

```
SPI_CS_LO is a macro that sets the SPI ChipSelect Low.
SPI_CS_HI is a macro that sets the SPI ChipSelect High.
```

In the spi.c file there are three interface functions:

```
spi_init()  Called by the higher level to initialise the
            interface

spi_rx_8()  Called by the higher level to read 8 bits of data
            from the interface

spi_tx_8()  Called by the higher level to transmit 8 bits of data
            through the interface.
```

If your ported driver accurately provides these five interface functions, then it will work.

4.4. Mounting the drive

The following code shows how to mount your DataFlash drive:

```
long memsize;
char *p1buffer;

memsize=fs_getmem_dataflashdrive(s_atmel_flash_fs_phy);
if (!memsize)
{
   /* configuration error */
}

p1buffer=(char*)malloc(memsize);

if (!p1buffer)
{
   /* Not enough memory to allocate */}
   return;
}

f_mountdrive(
   1,    /* drive number to use (1=B) */
   p1buffer,   memsize,
   fs_mount_dataflashdrive,
   s_atmel_flash_fs_phy);

/* The DataFlash Drive is ready for use! */
```

5. NAND Flash Driver

5.1. Overview

The NAND flash interface to the file system requires two files:

nflashdrv.c	- device independent flash control layer
K9F2816X0C.c	- physical chip controller

The **nflashdrv.c** module provides a single clean interface between the physical chip and the intermediate file system. This module gets information about the configuration of the underlying flash chip from the **K9F2816X0C.c** module and builds a controller based on that information. This module also does the wear-level control for the device.

> **Note**: Normally this module does not require modification. If modification is required it is strongly recommended that the developer contact CMX Systems about the specific requirements.

The **K9F2816X0C.c** module depends on the specific flash device and its configuration. Relevant data are the manufacturer, chip size, number of interface bits (8, 16 or 32), the number and arrangement of the chips (parallel or serial). All of these factors influence the code in this module.

The ***fs_phy_nand_k9f2816x0cxxx*** function is the key to understanding the interface between the specific physical driver and the file system. The structure returned by this call contains all configuration information about block usage required by the upper layers, as well as the set of interface function pointers to be used. The module provides the following interface functions to the **nflshdrv.c** module through the FS_FLASH structure:

NAND flash functions

- ***ReadFlash***
- ***EraseFlash***
- ***WriteFlash***
- ***VerifyFlash*** (optional)
- ***GetBlockSignature***
- ***CheckBadBlock***
- ***WriteVerifyPage***
- ***BlockCopy*** (only if static wear is used)

The only public function in this module is ***fs_phy_nand_K9F2816X0C***, which must be passed to ***f_mountdrive*** to initialize the physical driver. These functions are fully documented below.

All these functions require subroutine calls to do their work. After the function definitions descriptions are given for all the routines used in this module. These routines are documented for two K9F2816X0C Samsung chips in a parallel configuration. Implementation may vary for other devices. The routines are documented to give guidance as to how to implement this module.

5.2. Physical Device Usage

The developer must make some decisions about how to use the flash device. Basic considerations about all flash devices are: They are divided into a set of erasable blocks; it is possible to write only to an erased location; it is not possible to erase anything smaller than a block. Therefore, some complex management software is used.

Three types of blocks can be assigned to the device:

- Reserved blocks - for processes other than the file system; e.g., booting
- Descriptor blocks - to hold information about the structure of the file system, Such as wear, etc. By using a minimum of 2 descriptor blocks (and management software) the system is failsafe.
- File system blocks - for storing file information.

The sections below describe how to assign these.

5.2.1. Reserved blocks

You can reserve as many blocks as required from the physical device for private use. For example, if a particular physical device has 1024 erasable blocks and the user wants to reserve 256 blocks from the beginning for private use, the following statements could be used:

```
MAXBLOCK = 768    -  number of blocks for use by the file system
BLOCKSTART = 256  -  first file storage block
```

Note: The developer should take care not to access reserved blocks while the file system is accessing the device. Operations must be done atomically; i.e., a command must be completed on the device before another is started.

5.2.2. Descriptor Blocks

These blocks contain critical information about the file system: block allocation information, wear information and file/directory information. They are allocated automatically from the file system blocks.

The parameters that must be set up in the **fs_phy_nand_xxx** function are listed below:

descsize

This is the size of a descriptor block. Since all blocks are the same size on NAND flash devices it is the same as the block size.

seperatedir

Range 0 to 4. If this is set to a non-zero value the directory entries will be given blocks that are separate from the file system. The number specified in **separatedir** is the maximum number of separate blocks that will be allocated for directory entries. This allows a much larger number of files to be stored in the file system.

5.2.3. File System Blocks

The developer should allocate as many of these as required for file storage.

The parameters that must be set up in the ***fs_phy_nand_xxx*** function are listed below:

maxblock

This defines the number of erasable blocks available for file storage.

blockstart

This defines the logical number of the first of the blocks that may be used by the file system.

blocksize

This defines the size of the blocks to be used in the file storage area. It must be an erasable unit of the flash chip. All blocks in the file storage area must be the same size.

sectorsize

This defines the sector size. Each block is divided (by 2^n) into a number of sectors. This number is the smallest usable unit in the system and thus represents the minimum file storage area.

sectorperblock

This defines the number of sectors in a block. It must always be true that:

```
sectorperblock * sectorsize = blocksize
```

5.2.4. FS_NAND_RESERVEDBLOCK Definition

The file system needs to have a reserve of several blocks to ensure its smooth operation for all eventualities. The default value is 12. It must be at least 3 plus the number of separate directory blocks. Using a larger value than this is recommended to ensure that if bad blocks develop, there are others that can be used.

Note: At a certain point of usage all NAND blocks fail – once the number of failed blocks becomes too large to maintain a stable file system, then the system will return F_ERR_UNUSABLE and become a read-only file system.

5.3. Write Cache

The system allows a write cache to be defined for the driver. With the write cache, in most cases only changes to the descriptor block are stored in the flash device, thus improving the performance of the system (there are fewer erases and writes) and reducing wear on the system.

To use the write cache, ***WriteVerifyPage*** must be present. If this function does not exist then write caching cannot proceed.

Additionally the following parameters in the FS_FLASH structure must be set up in ***fs_phy_nand_xxx***:

```
cachedpagesize - should be equal to the page size of the device

cachedpagenum  - number of pages in the cache, which must equal
                 the number of pages in an erasable block.
```

If either of these is set to zero, write caching will not be used.

5.4. Maximum Files

The maximum number of file/directory entries that can be made on a file system is restricted. This number may be calculated from the formula:

MaxNum Entries = (Descsize - (maxblock*((sectorperblock*2) + 6)))/32

If more files are required (without using the **separatedir** setting) then either the sector size can be increased (creating more space in the descriptor blocks), or a larger descriptor block may be chosen. If fewer files are required then the sector size can be decreased or smaller descriptor blocks may be allocated.

If **separatedir** is used then the maximum number of file and directory entries is given by the formula:

MaxNum Entries = (Blocksize/32)***separatedir**

> **Note**: If files with long filenames are used the number of files that can be stored will be reduced.

5.5. Physical Layer Functions

5.5.1. fs_phy_nand_xxx

This is the first driver function called by the upper layer to retrieve information about the underlying physical driver. It can be used for initializing the flash device and detecting the flash type. The function must prepare the FS_FLASH structure with information so that the higher-level will know how to use this driver.

Format

```
int fs_phy_nand_xxx(FS_FLASH *flash)
```

Arguments

Argument	Description
flash	pointer to flash structure that must be filled

Return values

Return value	Description
0	success
else	error during initialization

Comments

This is the FS_FLASH structure that the module must set up:

```
typedef struct {
   long maxblock; /* max num of block that can be used */
      /* by the file system */
   long blocksize;   /*block size in bytes
   long sectorsize;  /*sector size to use
   long sectorperblock; /* sectors/block */
   long blockstart;     /*   the first physical block */
   long descsize;       /*block size in bytes */
   long descblock1;     /*not used for NAND */
   long descblock2;     /* not used for NAND */
   long separatedir;    /* directories use separate */
      /* block from FAT? */
   long cacheddescsize; /*not used for NAND */
   long cachedpagenum;  /*number of pages in cache */
   long cachedpagesize; /*size of pages in cache */
   FS_PHYREAD   ReadFlash;    /*read content fn ptr */
   FS_PHYERASE  EraseFlash;   /*erase a block fn ptr */
   FS_PHYWRITE  WriteFlash;   /*write content fn ptr */
   FS_PHYVERIFY VerifyFlash; /*verify content fn ptr */
   FS_PHYCHECK  CheckBadBlock; /* check if block is bad fn ptr */
   FS_PHYSIGN GetBlockSignature; /* get block signature data fn
       ptr */
   FS_PHYCACHE WriteVerifypage;     /* write and verify page */
   FS_PHYBLKCPY BlockCopy; /* HW/SW accelerated block copy */
   unsigned char *chkeraseblk;  /* buffer for preerasing blocks
      optional request to erase */
   unsigned char *erasedblk;   /* buffer for preerasing blocks
      optional set to erased */
} FS_FLASH;
```

5.5.2. ReadFlash

This function is called to read data from the flash device.

Format

```
int ReadFlash(void *data, long block, long blockrel, long
      datalen)
```

Arguments

Argument	Description
data	pointer to data storage location
block	zero based block number to be read
blockrel	relative position in block from where to start reading
datalen	length of data to be read

Return values

Return value	Description
0	success
any number	if there was any error during reading

Comments

Blockrel is a number that provides the location from which to start reading; it can range from 0 to block size.

Datalen is always less than block size and never points out from a given block, even if **blockrel** points into the middle of the block.

5.5.3. EraseFlash

Erase a block in flash.

Format

```
int EraseFlash(long block)
```

Arguments

Argument	Description
block	zero-based block number to be erased

Return values

Return value	Description
0	if successfully erased
any number	if there was any error during erasing

5.5.4. WriteFlash

Write data into the flash device.

Format

```
int WriteFlash(void *data, long block, long relsector, long size,
      long sdata)
```

Arguments

Argument	Description
data	points to source data to be written
block	zero-based block number where data are stored
relsector	zero based relative sector in block
size	length of data to be stored
sdata	block signature data

Return values

Return value	Description
0	if successfully written
any number	if there was any error during writing

5.5.5. VerifyFlash

This function verifies that a data range in the flash matches a data buffer. It is called after **WriteFlash** to verify written data against the original data.

Format

```
int VerifyFlash(void *data, long block, long relsector, long
      size, long sdata)
```

Arguments

Argument	Description
data	points to source data to be compared
block	zero-based block number where data are stored
relsector	zero based relative sector in block
size	length of data to be compare
sdata	block signature data

Return values

Return value	Description
0	if successfully verified and no different in device
any number	if there was any error during verifying

Comment

The verify routine is required only when verification is the desired method of ensuring that the device has been correctly written. To decide whether or not to use a verify routine, refer to the device datasheet. If, for example, ECC is being used and the guaranteed reliability is sufficient for your requirements, then the verify routine may be omitted. **This has a significant performance benefit.**

5.5.6. CheckBadBlock

This function is called at file system initialization to determine which blocks are bad. The flash device may contain invalid blocks; this function is called to register them so that the file system does not use them. The higher level will call this function for all used blocks. The method used to check for bad blocks is device dependent.

Format

```
int CheckBadBlock(long block)
```

Arguments

Argument	Description
block	number of block to be checked

Return values

Return value	Description
0	block is useable
1	block is BAD or INVALID

5.5.7. GetBlockSignature

This function is called from a higher level to get the previously stored block signature data set by ***WriteFlash()***.

Format

```
long GetBlockSignature(long block)
```

Arguments

Argument	Description
block	number of target block

Return values

Return value	Description
value	signature data

5.5.8. WriteVerifyPage

This function verifies that a page of data within the flash matches a buffer containing the written data. It is called after the write caching mechanism writes a page of data to the flash.

Format

```
int WriteVerifyPage(void *data, long block, long page, long
      pagenum, long sdata)
```

Arguments

Argument	Description
data	pointer to data to be written and verified
block	which block needs to be checked
page	start page number in block
pagenum	number of pages to be written
sdata	signature data for block

Return values

Return value	Description
0	success
else	failed

Comment

The verify routine is required only when verification is the desired method of ensuring that the device has been correctly written. To decide whether or not to use a verify routine, refer to the device datasheet. If, for example, ECC is being used and the guaranteed reliability is sufficient for your requirements, then the verify routine may be omitted. **This has a significant performance benefit.**

5.5.9. BlockCopy

This function copies one block to another block. It is called only if static wear leveling is being used. This routine should be implemented to use any features that may be used to accelerate a block-to-block copy operation. Many devices have features that help reduce CPU load and improve system performance. See the Static Wear section for further details.

Format

```
int BlockCopy(long destblock, long soublock)
```

Arguments

Argument	Description
destblock	block number to copy to
soublock	block number to copy from

Return values

Return value	Description
0	success
else	failed

5.6. Subroutine Descriptions and Notes for Sample Driver

This section contains a complete list of subroutines, with descriptions of their functionalities. It includes notes for porting to particular hardware designs.

NANDcmd(cmd: long)

Send a command to NAND flash

NANDaddr(addr: long)

Send an address to NAND flash

NANDwaitrb()

Wait until RB (ready/busy) goes high on NAND flash

ReadPage(pagenum: long)

Send command sequence to read a page
Read whole page data and calculate ECC
Get saved ECC from NAND flash spare area
If ECC calculation is needed do ECC checking

WritePage(data: ptr, pagenum: long, size: long)

Copy original data into a temporary buffer (this buffer is 32-bit aligned)
Send command sequence to NAND flash for programming a page
Program a whole page and calculate ECC
Write ECC into NAND flash spare area
Check if programming was successful; if not return with error

ReadFlash(data: ptr, block: long, blockrel: long, datalen: long)

Calculate **pagenum**
Find starting page from **blockrel**
ReadPage(**pagenum**)
Check if data needs to be copied (e.g. for alignment reasons) and copy if required.
ReadPage(**pagenum**) until **datalen**=0

EraseFlash(block: long)

Calculate **pagenum**
Send Command sequence to NAND flash erase block
Wait until erasing is finished
Check if erase was successful; if not return with error

WriteFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

Calculate **pagenum**
WritePage(**pagenum**++) until size=0 or any error
Signal error or return with successfully written

VerifyFlash(data: ptr, block: long, relsector: long, len: long, sdata: long)

Calculate **pagenum**
ReadPage(**pagenum**++) until **len**=0
Compare pages with original data; if any differences return with error

CheckBadBlock(block: long)

Determine whether or not given block is bad
Calculate **pagenum**
Send read spare area command to NAND flash
Check 6th word; if it's not 0xffffffff return with error; else return 0 (OK)

GetBlockSignature(block: long)

Read signature data from block

fs_phy_nand_K9F2816X0C (flash: struct)

Set function pointers for driver
Get device ID and manufacture ID from NAND flash
Compare all supported devices/manufacturers and fill flash structure with corresponding data (size, sectors, block information)
If device not found return with error

6. RAM Driver

Implementing a RAM driver for the file system is simple. There is no physical driver associated with the RAM driver.

1. Include the **ramdrv_s.c** and **ramdrv_s.h** files in your file system build. This ensures it can be mounted.

2. After ***f_init*** has been called, call ***f_mountdrive*** with a pointer to the memory area and the size of that area to be used for the driver.

```
#define RAM_DRIVE_SIZE 0x1000000

void main(void)
{
   f_init();         /* initialize the file system */
   f_enterFS();      /* get task access to the file system */

   /* mount first drive - A */
   f_mountdrive(
      0,          /* specifies drive 'A' */
      malloc (RAM_DRIVE_SIZE),/* get required buffer pointer */
      RAM_DRIVE_SIZE,      /* size of RAM drive to be used */
      fs_mount_ramdrive,   /*ramdrive    mount    function    (in
                           ramdrv.c) */
      0        /* no physical */
   )
}
```

The RAM drive may now be used as a standard drive.

In **ramdrv_s.c** there is a #define (MEMCPY_LONG) that can be set to 1 if the system supports 32-bit memory access. This accelerates memory access time and results in better performance.

7. File System Test

Two test suites are provided for exercising the file system, the flash drivers and ensuring that all are working correctly.

Both test programs require the functions defined and implemented (as samples) in **test-port_ram_s.c.** The functions must be ported according to the developer's system. See the comments and simple code for reference.

7.1. File System Test

Most functionality of the file system is exercised with this program, including file read/write/append/seek/file content, directories and file manipulation functions. To use the test program include **test_s.c** and **test_s.h** in your test project. Then

```
void f_dotest(void)
```

must be called to execute the test code.

7.2. Flash Driver Test

This code is designed to test your ported flash driver in isolation to ensure that it is ported correctly and is stable.

To use this test program include **testdrv_s.c** and **testdrv_s.h** in your test project.

Then

```
void f_dotestdrv(FS_PHYGETID phyfunc)
```

must be called to execute the test code.

Errors in the execution of this test indicate that there is an error in the implementation of the driver. Contact support@cmx.com if you need further advice.