HCC-Embedded

Embedded Flash File System

Implementation Guide

Version 1.91

All rights reserved. This document and the associated software are the sole property of HCC-Embedded Kft. Reproduction or duplication by any means of any portion of this document without the prior written consent of HCC-Embedded Kft. is expressly forbidden.

HCC-Embedded Kft. reserves the right to make changes to this document and to the related software at any time and without notice. The information in this document has been carefully checked for its accuracy; however, HCC-Embedded Kft. makes no warranty relating to the correctness of this document.

0 Contents

1 System Overview

Summary

EFFS is a package of source code and documentation designed for flash file system development on 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
- Unicode16 support

NOR Flash Support
- Wear-Leveling
- Bad Block Handling
- All known device types easily ported
- Sample drivers 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
- All known device types easily ported
- Sample driver with porting description

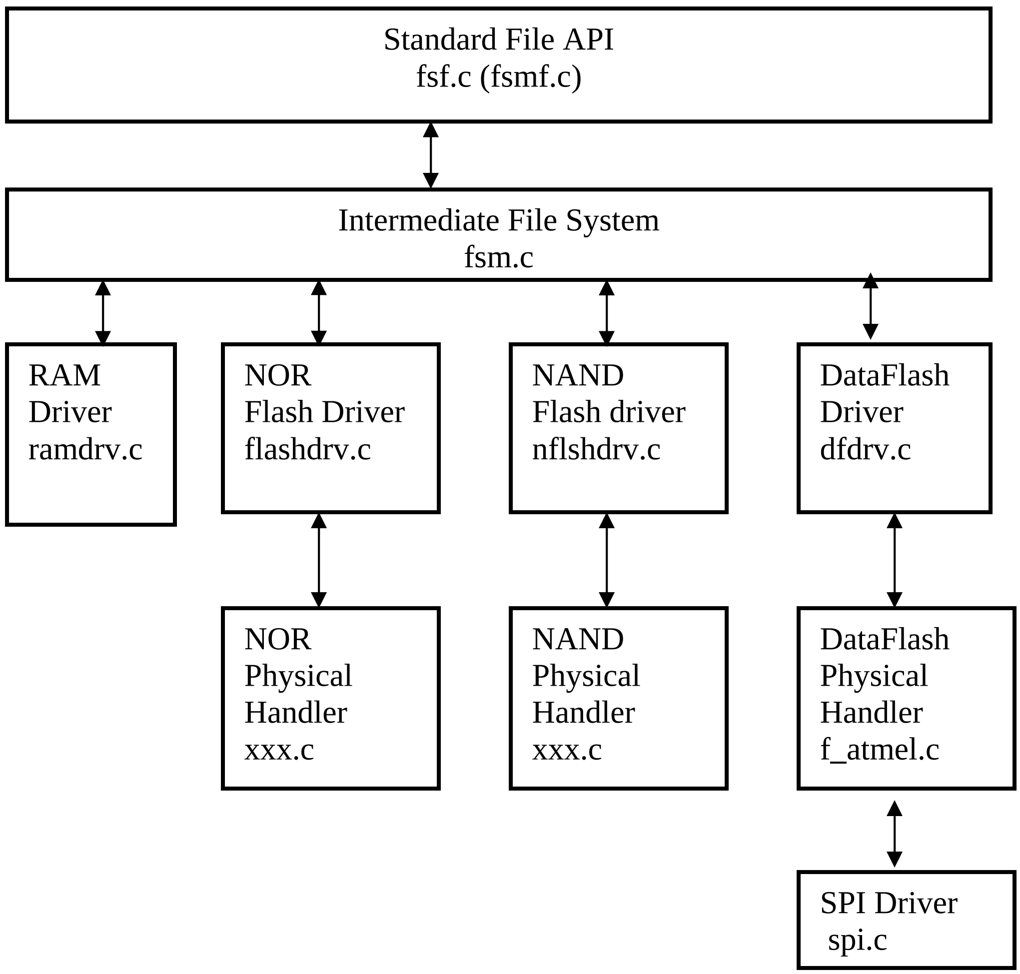
Target Audience

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

HCC-Embedded offers hardware and firmware development consultancy to assist developers with the implementation of a flash file system.

System Structure/Source Code

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



System Source File List

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

/src/common/

fsf.c - ffs Standard API
fsf.h - ffs Standard API header
fsmf.c - ffs Standard API Multi-thread wrapper
fsmf.h - ffs Standard API Multi-thread wrapper header
fsm.c - ffs intermediate layer
fsm.h - ffs intermediate layer header
port_s.c - functions to be ported
port_s.h - header file for port functions
udefs.h - user definitions header file

/src/ram/

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

/src/nor/

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/

dfdrv.c - DataFlash Generic Driver source
dfdrv.h - DataFlash Generic Handler header

/src/dflash/phy/atmel/

f_atmel.c – DataFlash physical handler
f_atmel.h – DataFlash physical handler header
spi.c - SPI comuniation sample driver
spi.h - SPI communicatin header file

/src/nand/

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

/src/nand/phy/samsung/

K9F2816X0C.c	- NAND flash physical handler
K9F2816X0C.h	- NAND flash physical handler header

/src/test/

test.c - test program source for exercising the file system
test.h - header file for test program
main.c - sample source file for running the test program

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.

What is NOR and NAND Flash?

The EFFS has been designed to allow the easy integration of all standard flash devices with the file system but what are these devices?

Flash devices have certain basic properties in common:

- They are designed for the non-volatile storage of code/data

- To write to an area it must be erased first - more precisely it is only possible to program a 1 to a 0. To change a 0 to a 1 an erase operation must be performed

- They are all divided into erase units (blocks) such that to erase any part a whole block must be erased

- They all wear-out after a number of erase cycles. This number of erase cycles guaranteed varies between chip types but an important feature of any file system using flash is wear management - the system seeks not to over use any one block.

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

NOR Flash

NOR flash has been the cornerstone of non-volatile memory in embedded systems for many years. Their basic characteristics are that they store data in a non-volatile way and importantly 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 slow such that even if quite small mounts of data are written an erase may be required causing a delay of as much as 2 seconds. Careful design of the file system has ensured that the number of occurrences of this is minimized but in certain cases it is not avoidable.

NAND/AND Flash

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

- They can store approximately 4 times more data than NOR technology for a similar price.

- They have much faster erase and write times making them ideal for applications which 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 - 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" in them which can never be used.

- Bits may flip unexpectedly (don't panic! - see below)

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

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

Within the NAND flash driver is contained the necessary spare area management and fast ECC algorithm.

NOR/NAND Summary

The following table summarizes the differences between NOR and NAND flash types - the entries given are indicative and subject to change between different parts and with time:

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

1. Blocks on NAND flash devices are normally smaller than blocks on NOR flash devices. Since an erase must precede writing to an area the smaller block size is generally beneficial to a file systems' performance.
2. The page size for NAND flash devices is typically 512 or 2048 bytes. Because file system access to the physical device is only in sectors the page access times are the most important when looking at 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.

Reentrancy

Certain sections of code must be protected from reentrancy - it is not a good idea to allow a user to start renaming a file just as another user is deleting it. Reentrancy, however, is not an issue on systems that can guarantee that at most one application will access the file system at one time.

Mutex Functions

If reentrancy is required as described in the previous section then the following functions in **port_s.c** must be implemented **–** normally provided by the host RTOS:

fs_mutex_create() **–** called at volume initialization
fs_mutex_delete() **–** called at volume deletion
fs_mutex_get() – called when a mutex is required
fs_mutex_put() – called when the mutex is released

Note: If the EFFS-CAPI is used (i.e. FS_CAPI_USED is defined in udefs.h) then these mutex functions will be replaced by those of the CAPI. Consult the EFFS-CAPI guide for further information

Maximum Tasks and CWD

If more than a single task is allowed to access the file system then reentrancy and maintenance of the current working directory must be considered.

Reentrancy is handled on a per volume basis and is documented in the sections above.

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** which can be changed by any user. This is maintained on a per volume basis.

An additional option has been provided which enables the file system to keep track of the **cwd** on a per calling task basis. To use this option the developer must take the following steps:

1. Set **`FS_MAXTASK`** in **udefs.h** to the maximum number of tasks that can simultaneously maintain access to the file system. This effectively creates a table of cwds 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 application using the file system calls `fs_releaseFS()` with its unique identifier to free that table entry for use by other applications.

Once this is done each caller will be logged as it acquires the semaphore, and a current working directory will be associated with it. The caller must release this when it has finished using the file system e.g. when the calling task is terminated. This frees the entry for other tasks to use.

Note: If the EFFS-CAPI is used (i.e. FS_CAPI_USED is defined in udefs.h) then the fn_gettaskID() function will be replaced by that in the CAPI. Consult the EFFS-CAPI guide for further information

Implementing Drivers

The driver design has been done to achieve 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 which does the translation between the driver and the physical flash hardware.

A detailed description of how this is implemented for NOR and NAND flash is contained in later sections of this manual.

Generally only the physical handler needs to be modified when the hardware configuration changes (different chip type, 1/2/4 devices in parallel etc). HCC-Embedded has a range of physical handlers available to make the porting process as simple as possible. HCC-Embedded also do specific porting work as required.

System Requirements

The 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 the system to work at its best certain porting work should be done as outlined below. This is a very straightforward task for an experienced engineer.

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 scheduling other operations while waiting for them to complete (e.g. a NOR flash erase is typically 2 seconds and a NAND flash erase 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 be made 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 be made to use the event generation mechanism of the host system so that other operations can be performed while waiting.

Real Time Clock

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

```
unsigned short fs_gettime(void)
unsigned short fs_getdate(void)
```

This function by default enters zeroes into these fields. When porting to a system with a real time clock, this 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 developers replacement function each time a file is created or closed.

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 their system.

These buffers vary in size depending on the precise chips being used and their configuration. For further information see description of ***fs_mountdrive*** function and the ***fs_getmem_xxx*** functions in the relevant driver sections.

Stack Requirements

The 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 calling file system functions. Typically calls to the file system will use <2Kbytes of stack.

Memcpy and Memset

The system includes ***memcpy*** and ***memset*** functions which are provided as simple byte copy routines. To get best performance from the target platform the developer should replace these routines 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.

Power Fail Safety

The flash file system is entirely power fail safe. The system may be stopped at any point and restarted and no data will be lost - the previous completed state of the file system will be restored.

When a file is closed its data is automatically flushed to the file system. Until this close takes place the file is preserved in its previous state. The user may also use the ***fs_flush*** command to write the current state of the file to the media and thus updating its failsafe state.

Long Filenames

The file system supports file names of almost unlimited length. The 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 the **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 uses 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.

Multiple Volumes

The file system supports multiple volumes. Each volume must have its own driver routine which normally then has its own physical routine (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. (E.g. if only RAM drive is used set the value to 1, if RAM drive and NOR flash then set this value to 2, etc).

Volume letters are assigned by passing a parameter in the ***fs_mountdrive*** function.

Multiple Open Files in a Volume

The file system 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 which determines the number of files that the file system allows to be opened simultaneously on that volume at any particular time.
For each file that may be allowed to be opened simultaneously an array must be allocated which contains a sector size buffer. Thus, increasing MAXFILES for a particular volume increases the RAM required by the system.

Static Wear

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

The file system uses a process called dynamic wear to allocate the least use blocks from those available. However, in systems where there are large areas of static data (e.g. the executable binary for the system) then the areas where this is stored may be written only once leaving a relatively small section of the device to handle the much more heavily used files.

For this reason a process called static wear 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 and if this difference is greater than a defined threshold (FS_STATIC_DISTANCE) then these two blocks will be exchanged in the system.

To use the static functionality the files fstaticw.c and fstaticw.h must be included in your project. In the header file two defines should be set:

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

FS_STATIC_PERIOD - this specifies how often this function will actually attempt to do a wear. This may be used in systems where ***fs_staticwear*** is called very frequently to reduce the number of times that the function will be executed. This reduces unnecessary checking of the system. If you always know that the system is going to be idle when ***fs_staticwear*** is called then this may be set to 1 so that it is always executed – for instance 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 actually execute this less frequently.

When the static wear function is executing the file system is not accessible. The length of time the static wear function takes is dependent 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 to function an additional driver function, ***BlockCopy***, must also be provided. See driver sections for information as to how to implement this function. It is important to provide a highly optimized version of this, preferably using any special copy functions specific to the target chip, to achieve the best system performance and least disruption.

Do I need static wear?

In many cases this is an unnecessary overhead – this can only be assessed by looking at how your product is to be used and considering the specification of your target devices. Many devices have up to 1 million erase/write cycles per block guaranteed and in many applications this number will not be reached in the lifetime of the product.

When should I do static wear?

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

Getting Started

To get your development started as efficiently as possible we recommend the following steps:

1. Build the file system using just the standard API (**fsf.c**), the intermediate file system (**fsm.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 the specific flash devices functionality.
3. Now add the next volume to the system either a NOR drive or 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 create a driver meeting your specific needs based on the available sample drivers.
   For DataFlash chips:
      Read Section 4 "Atmel DataFlash Driver" carefully and create a driver meeting your specific needs based on the available sample driver.
   For NAND chips:
      Read Section 5 "NAND Flash Driver" carefully and create a driver meeting your specific needs based on the available sample drivers.
5. Add new volumes by repeating steps 3 and 4.

2 File API

File System Functions

Common functions

- ***fs_getversion***
- ***fs_init***
- ***fs_mountdrive***
- ***fs_format***
- ***fs_getfreespace***
- ***fs_staticwear***

Drive\Directory handler functions

- ***fs_getdrive***
- ***fs_chdrive***
- ***fs_getcwd***
- ***fs_wgetcwd***
- ***fs_getdcwd***
- ***fs_wgetdcwd***
- ***fs_mkdir***
- ***fs_wmkdir***
- ***fs_chdir***
- ***fs_wchdir***
- ***fs_rmdir***
- ***fs_wrmdir***

File functions

- ***fs_rename***
- ***fs_wrename***
- ***fs_move***
- ***fs_wmove***
- ***fs_delete***
- ***fs_wdelete***
- ***fs_filelength***
- ***fs_wfilelength***
- ***fs_findfirst***
- ***fs_wfindfirst***
- ***fs_findnext***
- ***fs_settimedate***
- ***fs_wsettimedate***
- ***fs_gettimedate***
- ***fs_gettimedate***
- ***fs_setpermission***
- ***fs_wsetpermission***
- ***fs_getpermission***
- ***fs_wgetpermission***
- ***fs_truncate***
- ***fs_wtruncate***

Read/Write functions

- ***fs_open***
- ***fs_wopen***
- ***fs_close***
- ***fs_flush***
- ***fs_write***
- ***fs_read***
- ***fs_seek***
- ***fs_eof***
- ***fs_rewind***
- ***fs_putc***
- ***fs_getc***
- ***fs_seteof***
- ***fs_tell***

fs_getversion

This function is used to retrieve file system version information.

Format

```
char * fs_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",fs_getversion());
}
```

fs_init

This function initializes file system. This function must be called once to initialize file system before using any other file system function.

Format

```
void fs_init(void)
```

Arguments

None

Return values

None

Example

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

See also

```
fs_mountdrive, fs_format
```

fs_mountdrive

This is used to mount and map a new drive.

This function must be called with five parameters:

drivenum

Number of the drive to be mounted where 0 is drive 'A', 1 is drive 'B' etc. This has the maximum value of (FS_MAXVOLUME-1) in **fsm.h**.

buffer

This is a pointer for a buffer area to be used by the generic driver. Its size is dependent upon 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 this physical driver. The caller must then allocate this amount of memory and pass its pointer and size to the ***fs_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 this physical driver. The caller must then allocate this amount of memory and pass its pointer and size to the ***fs_mountdrive*** function. See example code below.

buffsize

This is the size of the allocated buffer being passed to the mount function.

mountfunc

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

The ***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 which 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 fs_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;
  fs_init();

  fs_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 */
       }
       fs_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_K9F28
       16X0C);
       if (!memsize) {
            /* nand flash is not identified, */
       }
       p2buffer=(char*)malloc(memsize);
       if (!p2buffer) {
            /* Not enough memory to allocate */
       }
       fs_mountdrive(
            2,
            p2buffer,
            memsize,
            fs_mount_nandflashdrive,
            fs_phy_nand_K9F2816X0C);
/* Drive C will be NAND flash drive with */
/* Samsung physical                      */
   }
```

See also

`fs_init, fs_format`

fs_format

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

Format

```
int fs_format(int drivenum)
```

Arguments

Argument	Description
drivenum	which drive needs to be formatted

Return values

Return value	Description
FS_NOERR	drive successfully formatted
FS_INVALIDDRIVE	if drive does not exist
FS_BUSY	if there is any file open
FS_NOTFORMATTED	if drive cannot be formatted

Example

```
char buffer[0x30000];

void myinitfs(void) {
  int ret;
  fs_init();
  ret=fs_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=fs_format(0); /* format drive A */
       if (ret==FS_NOTERR) return; /* formatted */
  }
initializationfailed:
  }
```

See also

```
fs_init, fs_mountdrive
```

fs_getfreespace

This function fills a user allocated structure with information about the usage of the volume specified. The information returned is the total size of the drive, the free space on the drive, the used space on the drive and the bad space on the drive.

Format

```
int fs_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
FS_NOERR	Success
else	Error code

Example

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

  ret = fs_getfreespace(fs_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);
}
```

fs_staticwear

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

Read “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
FS_NOERR	Success
else	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);
}
```

fs_mkdir

Make a new directory.

Format

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

Arguments

Argument	Description
dirname	new directory name to create

Return values

Return value	Description
FS_NOERR	new directory name created successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_INVALIDDIR	invalid path
FS_DUPLICATED	entry already exists
FS_NOMOREENTRY	directory is full

Example

```
void myfunc(void) {
  .
  .
  fs_mkdir(“subfolder”);   /* creating directory */
  fs_mkdir(“subfolder/sub1”);
  fs_mkdir(“subfolder/sub2”);
  fs_mkdir(“a:/subfolder/sub3”
  .
  .
}
```

See also

```
fs_chdir, fs_rmdir
```

fs_wmkdir

Make a new directory with Unicode16 name.

Format

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

Arguments

Argument	Description
dirname	new Unicode16 directory name to create

Return values

Return value	Description
FS_NOERR	new directory name created successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_INVALIDDIR	invalid path
FS_DUPLICATED	entry already exists
FS_NOMOREENTRY	directory is full

Example

```
void myfunc(void) {
  .
  .
  fs_wmkdir(“subfolder”);  /* creating directory */
  fs_wmkdir(“subfolder/sub1”);
  fs_wmkdir(“subfolder/sub2”);
  fs_wmkdir(“a:/subfolder/sub3”
  .
  .
}
```

See also

```
fs_wchdir, fs_wrmdir
```

fs_chdir

Change current working directory

Format

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

Arguments

Argument	Description
dirname	new directory name to change

Return values

Return value	Description
FS_NOERR	directory has been changed successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	path not found

Example

```
void myfunc(void) {
  .
  .
  fs_mkdir(“subfolder”);
  fs_chdir(“subfolder”);   /* change directory */
  fs_mkdir(“sub2”);
  fs_chdir(“..”);     /* go to upward */
  fs_chdir(“subfolder/sub2”);
                      /* goto into sub2 dir */
  .
  .
}
```

See also

fs_mkdir, fs_rmdir, fs_getcwd, fs_getdcwd

fs_wchdir

Change current working directory with Unicode16 name

Format

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

Arguments

Argument	Description
dirname	new Unicode16 directory name to change

Return values

Return value	Description
FS_NOERR	directory has been changed successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	path not found

Example

```
void myfunc(void) {
  .
  .
  fs_wmkdir(“subfolder”);
  fs_wchdir(“subfolder”);  /* change directory */
  fs_wmkdir(“sub2”);
  fs_wchdir(“..”);    /* go to upward */
  fs_wchdir(“subfolder/sub2”);
                    /* goto into sub2 dir */
  .
  .
}
```

See also

fs_wmkdir, fs_wrmdir, fs_wgetcwd, fs_wgetdcwd

fs_rmdir

Remove directory. Directory has to be empty when it is removed, otherwise returns with error code without removing.

Format

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

Arguments

Argument	Description
dirname	directory name to remove

Return values

Return value	Description
FS_NOERR	directory name is removed successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_NOTFOUND	directory not found
FS_INVALIDDIR	directory name is not a directory
FS_NOTEMPTY	directory not empty

Example

```
void myfunc(void) {
  .
  .
  fs_mkdir(“subfolder”); /* creating directories */
  fs_mkdir(“subfolder/sub1”);
  .
  . doing some work
  .
  fs_rmdir(“subfolder/sub1”);
  fs_rmdir(“subfolder”); /* removes directory */
  .
  .
}
```

See also

```
fs_mkdir, fs_chdir
```

fs_wrmdir

Remove Unicode16 directory. Directory has to be empty when it is removed, otherwise returns with error code without removing.

Format

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

Arguments

Argument	Description
dirname	Unicode16 directory name to remove

Return values

Return value	Description
FS_NOERR	directory name is removed successfully
FS_INVALIDNAME	directory name contains invalid characters
FS_NOTFOUND	directory not found
FS_INVALIDDIR	directory name is not a directory
FS_NOTEMPTY	directory not empty

Example

```
void myfunc(void) {
  .
  .
  fs_wmkdir(“subfolder”); /* creating directories */
  fs_wmkdir(“subfolder/sub1”);
  .
  . doing some work
  .
  fs_wrmdir(“subfolder/sub1”);
  fs_wmdir(“subfolder”); /* removes directory */
  .
  .
}
```

See also

```
fs_wmkdir, fs_wchdir
```

fs_getdrive

Get current drive number

Format

```
int fs_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=fs_getdrive();
  .
  .
}
```

See also

```
fs_chdrive
```

fs_chdrive

Change current drive.

Format

```
int fs_chdrive(int drivenum)
```

Arguments

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

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDDRIVE	drive number is invalid

Example

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

See also

fs_getdrive

fs_getcwd

Get current working folder on current drive.

Format

```
int fs_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
FS_NOERR	Success
FS_INVALIDDRIVE	Current drive is invalid

Example

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

  if (!fs_getcwd(buffer, FS_MAXPATH)) {
       printf (“current directory is %s”,buffer);
  }
  else {
       printf (“Drive Error”)
  }
}
```

See also

```
fs_chdir, fs_getdcwd
```

fs_wgetcwd

Get current working folder on current drive.

Format

```
int fs_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
FS_NOERR	Success
FS_INVALIDDRIVE	Current drive is invalid

Example

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

  if (!fs_wgetcwd(buffer, FS_MAXPATH)) {
       wprintf (“current directory is %s”,buffer);
  }
  else {
       wprintf (“Drive Error”)
  }
}
```

See also

fs_wchdir, fs_wgetdcwd

fs_getdcwd

Get current working folder on selected drive.

Format

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

Arguments

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

Return values

Return value	Description
FS_NOERR	Success
FS_INVALIDDRIVE	Current drive is invalid

Example

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

  if (!fs_getcwd(drivenum,buffer, FS_MAXPATH)) {
       printf (“current directory is %s”,buffer);
       printf (“on drive %c”,drivenum+’A’);
  }
  else {
       printf (“Drive Error”)
  }
}
```

See also

```
fs_chdir, fs_getcwd
```

fs_wgetdcwd

Get current working folder on selected drive.

Format

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

Arguments

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

Return values

Return value	Description
FS_NOERR	Success
FS_INVALIDDRIVE	Current drive is invalid

Example

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

  if (!fs_wgetcwd(drivenum,buffer, FS_MAXPATH)) {
       wprintf (“current directory is %s”,buffer);
       wprintf (“on drive %c”,drivenum+’A’);
  }
  else {
       wprintf (“Drive Error”)
  }
}
```

See also

fs_wchdir, fs_wgetcwd

fs_rename

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

Format

```
int fs_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
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found
FS_BUSY	file is open for read or write
FS_DUPLICATED	name already exists

Example

```
void myfunc(void) {
  .
  .
  fs_rename (“oldfile.txt”,”newfile.txt”);
  fs_rename (“A:/subdir/oldfile.txt”,”newfile.txt”);
  .
  .
}
```

See also

```
fs_mkdir, fs_open, fs_move
```

fs_wrename

Rename a file or directory with unicode16 name. This function has been obsoleted by ***fs_wmove***.

Format

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

Arguments

Argument	Description
filename	unicode16 file or directory name with/without path
newname	new unicode16 name of file or directory

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found
FS_BUSY	file is open for read or write
FS_DUPLICATED	name already exists

Example

```
void myfunc(void) {
  .
  .
  fs_wrename (“oldfile.txt”,”newfile.txt”);
  fs_wrename (“A:/dir/oldfile.txt”,”newfile.txt”);
  .
  .
}
```

See also

```
fs_wmkdir, fs_wopen, fs_wmove
```

fs_move

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

Format

```
int fs_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
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found
FS_BUSY	file is open for read or write
FS_DUPLICATED	name already exists

Example

```
void myfunc(void) {
  .
  .
  fs_move (“oldfile.txt”,”newfile.txt”);
  fs_move (“A:/subdir/oldfile.txt”,
            “A:/newdir/oldfile.txt”);
  .
  .
}
```

See also

```
  fs_mkdir, fs_open, fs_rename
```

fs_wmove

Moves a file or directory with unicode16 name. The original is lost. This function obsoletes ***fs_wrename***. The source and target must be in the same volume.

Format

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

Arguments

Argument	Description
filename	unicode16 file or directory name with/without path
newname	new unicode16 name of file or directory

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found
FS_BUSY	file is open for read or write
FS_DUPLICATED	name already exists

Example

```
void myfunc(void) {
  .
  .
  fs_wmove (“oldfile.txt”,”newfile.txt”);
  fs_wmove (“A:/subdir/oldfile.txt”,
            “A:/newdir/oldfile.txt”);
  .
  .
}
```

See also

```
fs_wmkdir, fs_wopen, fs_wrename
```

fs_delete

Delete a file.

Format

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

Arguments

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

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file not found
FS_BUSY	file is open for read or write
FS_INVALIDDIR	file name is a directory name

Example

```
void myfunc(void) {
  .
  .
  fs_delete (“oldfile.txt”);
  fs_delete (“A:/subdir/oldfile.txt”);
  .
  .
}
```

See also

`fs_open`

fs_wdelete

Delete a file with unicode16 name.

Format

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

Arguments

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

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	filename contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file not found
FS_BUSY	file is open for read or write
FS_INVALIDDIR	file name is a directory name

Example

```
void myfunc(void) {
  .
  .
  fs_wdelete (“oldfile.txt”);
  fs_wdelete (“A:/subdir/oldfile.txt”);
  .
  .
}
```

See also

```
fs_wopen
```

fs_filelength

Get the length of a file.

Format

```
long fs_filelength (char *filename)
```

Arguments

Argument	Description
filename	file name with or without path

Return values

Return value	Description
filelength	length of file

Example

```
int myreadfunc(char *filename, char *buffer, long
buffsize) {
  FS_FILE *file=fs_open(filename,”r”);
  long size=fs_filelength(filename);
  if (!file) {
       printf (“%s Cannot be opened!”,filename);
       return 1;
  }
  if (size>buffsize) {
       printf (“Not enough memory!”);
       return 2;
  }

  fs_read(buffer,size,1,file);
  fs_close(file);

  return 0;
}
```

See also

```
fs_open
```

fs_wfilelength

Get the length of a file with unicode16 name.

Format

```
long fs_wfilelength (W_CHAR *filename)
```

Arguments

Argument	Description
filename	unicode16 file name with or without path

Return values

Return value	Description
filelength	length of file

Example

```
int myreadfunc(W_CHAR *filename, char *buffer, long
buffsize) {
  FS_FILE *file=fs_wopen(filename,”r”);
  long size=fs_wfilelength(filename);
  if (!file) {
       printf (“%s Cannot be opened!”,filename);
       return 1;
  }
  if (size>buffsize) {
       printf (“Not enough memory!”);
       return 2;
  }

  fs_read(buffer,size,1,file);
  fs_close(file);

  return 0;
}
```

See also

```
fs_wopen
```

fs_findfirst

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

Format

```
int fs_findfirst(const char *filename,FS_FIND
*find)
```

Arguments

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

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	file name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_INVALIDDIR	invalid path
FS_NOTFOUND	file not found

Example

```
void mydir(void) {
  FS_FIND find;
  if (!fs_findfirst("A:/subdir/*.*",&find)) {
       do {
            printf (“filename:%s”,find.filename);
            if (find.attr&FS_ATTR_DIR) {
                 printf (“ directory\n”);
            }
            else {
                 printf (“ size %d\n”,find.len);
            }
       } while (!fs_findnext(&find));
  }
}
```

See also

```
fs_findnext
```

fs_wfindfirst

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

Format

```
int fs_wfindfirst(const W_CHAR *filename,
FS_WFIND *find)
```

Arguments

Argument	Description
filename	unicode16 name of file to find
find	where to store find information

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	file name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_INVALIDDIR	invalid path
FS_NOTFOUND	file not found

Example

```
void mydir(void) {
  FS_WFIND find;
  if (!fs_wfindfirst("A:/subdir/*.*",&find)) {
       do {
            printf (“filename:%s”,find.filename);
            if (find.attr&FS_ATTR_DIR) {
                 printf (“ directory\n”);
            }
            else {
                 printf (“ size %d\n”,find.len);
            }
       } while (!fs_wfindnext(&find));
  }
}
```

See also

```
fs_wfindnext
```

fs_findnext

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

Format

```
int fs_findnext(FS_FIND *find)
```

Arguments

Argument	Description
find	find structure (from ***fs_findfirst***)

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file not found

Example

```
void mydir(void) {
  FS_FIND find;
  if (!fs_findfirst("A:/subdir/*.*",&find)) {
       do {
            printf (“filename:%s”,find.filename);
            if (find.attr&FS_ATTR_DIR) {
                 printf (“ directory\n”);
            }
            else {
                 printf (“ size %d\n”,find.len);
            }
       } while (!fs_findnext(&find));
  }
}
```

See also

fs_findfirst, fs_findfirst

fs_wfindnext

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

Format

```
int fs_wfindnext(FS_WFIND *find)
```

Arguments

Argument	Description
find	find structure (from ***fs_wfindfirst)***)

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file not found

Example

```
void mydir(void) {
  FS_WFIND find;
  if (!fs_wfindfirst("A:/subdir/*.*",&find)) {
       do {
            printf (“filename:%s”,find.filename);
            if (find.attr&FS_ATTR_DIR) {
                 printf (“ directory\n”);
            }
            else {
                 printf (“ size %d\n”,find.len);
            }
       } while (!fs_wfindnext(&find));
  }
}
```

See also

```
fs_wfindfirst, fs_wfindfirst
```

fs_settimedate

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

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 fs_settimedate(const char *filename,
                   unsigned short ctime,
                   unsigned 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
FS_NOERR	success
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_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. */
  fs_mkdir(“subfolder”);   /* creating directory */
  fs_settimedate(“subfolder”,ctime,cdate);
}
```

See also

```
fs_gettimedate
```

fs_wsettimedate

Set time and date on a file or on a directory with Unicode16 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 fs_settimedate(const W_CHAR *filename,
                       unsigned short ctime,
                       unsigned short cdate)
```

Arguments

Argument	Description
filename	unicde16 name of file
ctime	creation time of file or directory
cdate	creation date of file or directory

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_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. */
  fs_wmkdir("subfolder");  /* creating directory */
  fs_wsettimedate("subfolder",ctime,cdate);
}
```

See also

```
  fs_wgettimedate
```

fs_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 is simply two 16-bit numbers associated with the specified file which the developer is free to use as desired.

Format

```
int fs_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
FS_NOERR	success
FS_INVALIDNAME	file or directory name contains invalid characters
FS_NOTFOUND	file or directory not found

Example

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

fs_settimedate

fs_wgettimedate

Get time and date information from a file or directory with Unicode16 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 is simply two 16-bit numbers associated with the specified file which the developer is free to use as desired.

Format

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

Arguments

Argument	Description
filename	unicode16 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
FS_NOERR	success
FS_INVALIDNAME	file or directory name contains invalid characters
FS_NOTFOUND	file or directory not found

Example

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

fs_wsettimedate

fs_setpermission

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

Every file/directory in the file system has 32-bit field associated with it – this is known as the permission setting. This field is freely programmable by the developer and could, for instance, be used to create a user access system.

Format

```
int fs_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
FS_NOERR	success
FS_INVALIDNAME	file or directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found

Example

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

See also

```
  fs_getpermission
```

fs_wsetpermission

This sets the file or directory permission field associated with a file with Unicode16 name.

Every file/directory in the file system has 32-bit field associated with it – this is known as the permission setting. This field is freely programmable by the developer and could, for instance, be used to create a user access system.

Format

```
int fs_wsetpermission(const W_CHAR *filename,
                      unsigned long secure)
```

Arguments

Argument	Description
filename	Unicode16 name of target file
secure	32bit number to associate with filename

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	file or directory name contains invalid characters
FS_INVALIDDRIVE	drive does not exist
FS_NOTFOUND	file or directory not found

Example

```
void myfunc(void) {
  fs_mkdir(“subfolder”);   /* creating directory */
  fs_wsetpermission (“subfolder”,0x00ff0000);
}
```

See also

```
fs_wgetpermission
```

fs_getpermission

Retrieves file or directory permission field associated with a file.

Every file/directory in the file system has a 32-bit field associated with it - this is known as the permission setting. This field is freely programmable by the developer and could, for instance, be used to create a user access system.

Format

```
int fs_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
FS_NOERR	success
FS_INVALIDNAME	file or directory contains invalid characters
FS_NOTFOUND	file or directory not found

Example

```
void myfunc(void) {
  unsigned long secure;
  if (!fs_getpermission (“subfolder”,&secure)) {
       printf (“permission is: %d”,secure);
  }
  else {
       printf (“Permission cannot be retrieved!”);
  }
}
```

See also

```
fs_setpermission
```

fs_wgetpermission

Retrieves file or directory permission field associated with a file with Unicode16 name.

Every file/directory in the file system has a 32-bit field associated with it - this is known as the permission setting. This field is freely programmable by the developer and could, for instance, be used to create a user access system.

Format

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

Arguments

Argument	Description
filename	unicode16 name of target file
psecure	pointer to where to store permission

Return values

Return value	Description
FS_NOERR	success
FS_INVALIDNAME	file or directory contains invalid characters
FS_NOTFOUND	file or directory not found

Example

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

See also

```
fs_wsetpermission
```

fs_open

Open a file. The following open modes are allowed:

“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.

Nb. The system handles all files in binary mode. There is no text mode support.

Format

```
FS_FILE *fs_open(const char *filename,
                 const char *mode);
```

Arguments

Argument	Description
filename	target file
mode	open mode

Return values

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

Example

```
void myfunc(void) {
  FS_FILE *file;
  char c;
  file=fs_open("myfile.bin","r");
  if (!file) {
       printf ("File cannot be opened!");
       return;
  }
  fs_read(&c,1,1,file); /* read 1byte */
  printf ("'%c' is read from file",c);
  fs_close(file);
}
```

See also

fs_read, fs_write, fs_close,

fs_wopen

Open a file with Unicode16 filename. The following open modes are allowed:

"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.

Nb. The system handles all files in binary mode. There is no text mode support.

Format

```
FS_FILE *fs_wopen(const W_CHAR *filename,
                  const char *mode);
```

Arguments

Argument	Description
filename	unicode16 name of target file
mode	open mode

Return values

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

Example

```
void myfunc(void) {
  FS_FILE *file;
  char c;
  file=fs_wopen("myfile.bin","r");
  if (!file) {
       wprintf ("File cannot be opened!");
       return;
  }
  fs_read(&c,1,1,file); /* read 1byte */
  wprintf ("'%c' is read from file",c);
  fs_close(file);
}
```

See also

```
  fs_read, fs_write, fs_close
```

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

See also

`fs_open`

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

See also

`fs_wopen`

fs_close

Close a previously opened file.

Format

```
int fs_close(FS_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
FS_NOERR	success
FS_NOTOPEN	file not open
FS_INVALIDDRIVE	file handle points to invalid drive
FS_DRIVEERROR	Cannot be written into device

Example

```
void myfunc(void) {
  FS_FILE *file;
  char *string=”ABC”;
  file=fs_open(“myfile.bin”,”w”);
  if (!file) {
       printf (“File cannot be opened!”);
       return;
  }
  fs_write(string,3,1,file); /* write 3byte */
  if (!fs_close(file)) {
       printf (“File stored”);
  }
  else printf (“file close error”);
}
```

See also

fs_open, fs_read, fs_write

fs_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 closing and opening the file. Once this command has completed this new state of the file will be restored after a system failure.

Format

```
int fs_flush(FS_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

Return value	Description
FS_NOERR	success
FS_NOTOPEN	file not open
FS_INVALIDDRIVE	file handle points to invalid drive
FS_DRIVEERROR	Cannot be written into device

Example

```
void myfunc(void) {
  FS_FILE *file;
  char *string=”ABC”;
  file=fs_open(“myfile.bin”,”w”);
  if (!file) {
       printf (“File cannot be opened!”);
       return;
  }
  fs_write(string,3,1,file); /* write 3byte */
  if (!fs_flush(file)) {
       printf (“New data is now failsafe”);
  }
  else printf (“file flush error”);
}
```

See also

```
fs_open, fs_write, fs_close
```

fs_write

Write data into file at current position. File has to 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 and ***fs_flush*** or ***fs_close*** has been done on the file.

Format

```
long fs_write(const void *buf,
              long size,long size_st,
              FS_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) {
  FS_FILE *file;
  char *string=”ABC”;
  file=fs_open(“myfile.bin”,”w”);
  if (!file) {
       printf (“File cannot be opened!”);
       return;
  }
  if (fs_write(string,1,3,file)!=3)
  {    /* write 3bytes */
       printf (“not all items written”);
  }
  fs_close(file);
}
```

See also

fs_read, fs_open, fs_close, fs_flush

fs_read

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

Format

```
long fs_read(void *buf,
             long size,long size_st,
             FS_FILE *filehandle)
```

Arguments

Argument	Description
buf	buffer where to store data
size	size 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 read bytes

Example

```
int myreadfunc(char *filename, char *buffer, long
buffsize) {
  FS_FILE *file=fs_open(filename,”r”);
  long size=fs_filelength(filename);
  if (!file) {
       printf (“%s Cannot be opened!”,filename);
       return 1;
  }
  if (fs_read(buffer,1,size,file)!=size) {
       printf (“not all items read”);
  }
  fs_close(file);
  return 0;
}
```

See also

`fs_seek, fs_tell, fs_open, fs_close, fs_write`

fs_seek

Move read/write position in the file. Whence parameter could be one of:
FS_SEEK_CUR - Current position of file pointer
FS_SEEK_END - End of file
FS_SEEK_SET - Beginning of file
offset position is relative to whence.

Format

```
int fs_seek(FS_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
FS_NOERR	success
FS_NOTFORREAD	file not open for reading
FS_NOTUSEABLE	whence parameter is invalid
FS_DRIVEERROR	drive is not readable
FS_INVALIDDRIVE	invalid drive specified in file handle

Example

```
int myreadfunc(char *filename, char *buffer, long
buffsize) {
  FS_FILE *file=fs_open(filename,"r");
  fs_read(buffer,1,1,file); /* read 1 byte */
  fs_seek(file,0,SEEK_SET);
  fs_read(buffer,1,1,file);/*read the same 1 byte */
  fs_seek(file,-1,SEEK_END);
  fs_read(buffer,1,1,file); /* read last 1 byte */
  fs_close(file);
  return 0;
}
```

See also

fs_read, fs_tell

fs_tell

Tell the current file position in the target file.

Format

```
long fs_tell(FS_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) {
  FS_FILE *file=fs_open(filename,"r");
  printf ("Current position %d",fs_tell(file));
  fs_read(buffer,1,1,file); /* read 1 byte */
  printf ("Current position %d",fs_tell(file));
  fs_read(buffer,1,1,file); /* read 1 byte */
  printf ("Current position %d",fs_tell(file));
  fs_close(file);
  return 0;
}
```

See also

```
fs_seek, fs_read, fs_write, fs_open
```

fs_seteof

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

Format

```
int fs_seteof(F_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	handle of open target file

Return values

Return value	Description
0	Success
else	Failed – see error codes

Example

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

  fs_seek(file,position,SEEK_SET);

  if(fs_seteof(file))
       printf("Truncate Failed\n");

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

See also

```
fs_truncate, fs_write, fs_open
```

fs_eof

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

Format

```
int fs_eof(FS_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) {
  FS_FILE *file=fs_open(filename,"r");
  while (!fs_eof()) {
       if (!buffsize) break;
       buffsize--;
       fs_read(buffer++,1,1,file);
  }
  fs_close(file);
  return 0;
}
```

See also

```
fs_seek, fs_read, fs_write, fs_open
```

fs_rewind

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

Format

```
int fs_rewind(FS_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];
  FS_FILE *file=fs_open("myfile.bin",”r”);
  if (file) {
     fs_read(buffer,4,1,file);
     fs_rewind(file); /* rewind file pointer */
     fs_read(buffer2,4,1,file);
                      /* read from beginning */
     fs_close(file);
  }
  return 0;
}
```

See also

```
fs_seek, fs_read, fs_write, fs_open
```

fs_putc

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

Format

```
int fs_putc(int ch,FS_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) {
  FS_FILE *file=fs_open(filename,"w");
  while (num--) {
  int ch='A';
       if(ch!=(fs_putc(ch))
       {
            printf("fs_putc error!");
            break;
       }
  }
  fs_close(file);
  return 0;
}
```

See also

```
fs_seek, fs_read, fs_write, fs_open
```

fs_getc

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

Format

```
int fs_getc(FS_FILE *filehandle)
```

Arguments

Argument	Description
filehandle	file handle of target

Return values

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

Example

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

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

See also

fs_seek, fs_read, fs_write, fs_open, fs_eof

3 NOR Flash Driver

Physical Device Usage

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

The file system operates on a set of logical blocks that may be further divided into sectors. The physical driver has to 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.

and

2. Provides a mapping between the logical block numbers used by the file system to 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 use 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.

Reserved blocks

The developer can reserve as many blocks from the physical device as required for private usage. This is done simply by omitting those blocks from the ***GetBlockAddr*** function.

If the developer wants to access reserved blocks using the ***GetBlockAddr*** function then this may also be done by selecting the physical block numbers to be used and ensuring they are not used by those specified in the descriptor and file system usage below.

Note: Care should be taken in accessing reserved blocks and attention 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 a sensible approach to use only the file system or the reserved sectors at any one time. Otherwise careful understanding of the specific device used is required.

Descriptor Blocks

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

These blocks contain critical information about the file system, block allocation, wear information and file/directory information. At least two descriptor blocks must be included in the system, which can be erased independently. An optional descriptor write cache may be configured which 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 following definitions for parameters that must be set up in the NOR physical header file are listed below:

DESCBLOCKSIZE

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 - as long as it fits in a single physical block that is used only for this single purpose.

DESCBLOCKSTART

This is the logical number of the first descriptor 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. There must be at least two descriptor blocks defined.

DESCCACHE

This defines the descriptor write cache size. This number must be less than DESCBLOCKSIZE - the cache is allocated in the descriptor block. If set to zero the descriptor write cache method will not be used. Use of the descriptor write cache is an efficient method of updating the changes in the descriptor such that the whole descriptor need not be re-written - while still retaining the 100% power-fail safe characteristics of the system.

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

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

File System Blocks

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

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

MAXBLOCKS

This defines the number of erasable blocks available for file storage

BLOCKSTART

This defines the logical number of the first of these blocks 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. This must be an erasable unit of the flash chip. All blocks in the file storage area must be the same size. This maybe different from the DESCSIZE (see above) where the flash chip has different size erasable units available.

SECTORSIZE

This defines the sector size. Each block is divided into a number of sectors. This number 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 sector section below.

SECTORPERBLOCK

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

SECTORPERBLOCK = BLOCKSIZE/SECTORSIZE

Example 1

The target flash device (e.g. AM29LV160B - see 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 then 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

DESCBLOCKSIZE	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

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

```
{
     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));
}
```

Example 2

Using a flash device with 512*128K erasable blocks (e.g AM29LV2562M - see 29lv2562m.c file for reference). A minimum of two erasable blocks must be used for descriptors but 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 this large cache has two advantages in that the number of erases required 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- 509used)
DESCBLOCKSIZE	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

The code below shows possible modifications to the driver:

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

```
{
     return((block*BLOCKSIZE)+(relsector*SECTORSIZE));
}
```

Sectors and File Storage

The blocks of the file storage section of the file system are sub-divided into equal sized sectors. These sectors are the minimum write-able 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 to the system. An additional benefit of keeping the sector size small is that if small files are written many more can be written before a block erase is required.

e.g. if there is 1 sector per block then a block must be erased for every file but 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. **Use fsmem.exe!**

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 file 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 the developer should consider using larger descriptor blocks but this comes with a penalty that the erase time of the frequently used descriptor blocks will increase.

Note: HCC-Embedded provides an executable program (**/utils/fsmem.exe**) for calculating the capabilities of a particular file system on based on input configuration information.

Note: If files with longer names are used the total number of files that can be stored will be reduced.

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

Normally this module does not require modification. If modification is required it is strongly recommended that the developer contact HCC-Embedded about their requirements.

The **29lvxxx.c** module is dependent on the specific flash device used and its configuration – i.e. which manufacturer, what size is the chip, is it a 8/16/32 bit interface and are there several 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 (only required if static wear used)

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

All these functions are documented below. These functions then require subroutine calls to fulfill their function. After these 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. For any specific device the implementation may vary. The routines are documented to give guidance as to how to implement this module.

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 device/s used and the hardware design. A sample driver for an AMD29Lxx device is supplied for reference purposes.

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 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 which 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 */

} FS_FLASH;
```

ReadFlash

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

Format

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

Arguments

Argument	Description
data	pointer where to store data
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, which says the reading start position in block, could be a number from 0 to block size.

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

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

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 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 need to store
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

VerifyFlash

This function is called from higher level after ***WriteFlash*** to verify written data. 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 where to compare data
relsector	zero based relative sector number in block
size	length of data need to compare
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 – this depends on the particular flash chip in use and what is specified in the datasheet to guarantee that a program operation has completed successfully.

BlockCopy

This function copies one block to another block. This function is only called if static wear is being used. This routine should be implemented to use any features of the target device which may be used to accelerate a block to block copy operation. Many devices have features to support this which helps 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

Subroutine Descriptions and Notes for Sample Driver

This section contains a complete list of subroutines, describes their functionality and includes notes for porting these routines to a particular hardware design.

FS_FLASHBASE

This define specifies the base address for accessing the flash memory array. The value of this can only be determined from the hardware design. The 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 implement their own function here to remove write protection based on their 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 implement their own function here to set write protection based on their 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 returned.

This routine must be modified by the developer to return the correct block/sector addresses for the requested logical blocks as has 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 to that of the specific type of flash device being used. The sample program is that for an AM29xxxx series flash device.

***DataPoll*(addr: long, chk ushort)**

This is an AMD specific sub-routine for checking that data has 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 to that of the specific type of flash device being used. The sample program is that for an AM29xxxx series flash device.

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

This routine is called by the higher levels to write some 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 to that of the specific type of flash device being used. The sample program is that 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 to that of the specific type of flash device being used. The sample program is that 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 to that of the specific type of flash device being used. The sample program is that for an AM29xxxx series flash device.

***fs_phy_nor_29lvxxx* (flash: struct)**

Initialises internal functions to the flash structure
RemoveWriteProtect()
Getting device ID and manufacture ID from the flash
SetWriteProtect()
Compare all supported device/manufacture and fills flash structure with corresponding data (size, sectors, block information)
If device not found returns with error

***FnWriteWord* (base: ptr, addr: long, data: ushort)**

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

Pre-Erase and Erase Suspend/Resume

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

By pre-erasing dirty blocks the performance of your system can be greatly improved.

Pre-erase can only be done on devices which have commands for suspending and resuming the erase operation. Since erase operations can take several seconds on NOR some NOR flash devices this option can greatly reduce system latencies. The host system must also 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 as to how to implement this logic. The following describes how to implement this feature based on this driver sample.

The general operation is that file system calls will operate at higher priority than the LowFlashErase task – otherwise the file system operation will have to wait on all blocks that can be pre-erased to be erased. Each time a read or write on the flash is required any active erase will be suspended, the operation will be completed and the erase operation will be resumed.

In the sample driver the function *LowFlashErase()* should be called regularly by a task of lower priority 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:

SuspendErase() – sends a command to the flash to suspend the erase operation
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 doing the operation and then call *ResumeErase()* and put the *gl_mutex* once the operation is complete.

4 Atmel DataFlash™ Driver

These devices have their own particular design characteristics which make it important to design a driver specifically for them to be able to use them reliably.

The main features are:

- Data is 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 a sector is written to 10,000 times 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.

To implement this interface is straightforward. Include the dfdrv.c, f_atmel.c and spi.c files and their headers in your project and follow the sections below.

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.

Firstly define your device type in **f_atmel.h** from those listed:

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

If your device is not one of those listed contact support@hcc-embedded.com

This definition will automatically set the following definitions to their default value for that 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
```

These are tested configurations using the whole of the 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 size of sectors to be used by the file system. Should always be a multiple of 8 pages.

`F_ATMEL_BLOCKSIZE` The size of blocks to be used by the file system. Should always be a power of 2 multiple of the sector size.

`F_ATMEL_CACHED_DESC` Number of cache descriptors used.

`F_ATMEL_NO_OF_DESC_BLOCKS` Number of descriptor blocks used.

The following definitions may be used to reserve sectors of the data use for use outside the file system:

`F_ATMEL_RESERVE_FROM_SECTOR` This sets the number of sectors at the end of the flash which will be excluded from management by the file system.

`F_ATMEL_MANAGEMENT_SECTOR` This specifies the first sector of the device that will be used for management of the file system. All sectors before this are free for alternate use. It is recommended to use a large sector for this management sector to benefit the overall wear distribution of the system.

Porting the SPI interface

The physical interface to the DataFlash is through SPI. A simple, sample SPI driver is provided. Because all platforms implement their SPI interface in different ways thi diver 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 which sets the SPI ChipSelect Low.
`SPI_CS_HI` is a macro which 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.

Mounting the drive

The following code shows how to mount your DataFlash drive:

```
long memsize;
char *p1buffer;

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

p1buffer=(char*)malloc(memsize);

if (!p1buffer) {/* Not enough memory to allocate */}
fs_mountdrive(
     1,        /* drive number to use (1=B) */
     p1buffer,
     memsize,
     fs_mount_dataflashdrive,
     f_atmel_flash_fs_phy);

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

5 NAND Flash Driver

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 for the physical chip to 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.

Normally this module does not require modification. If modification is required it is strongly recommended that the developer contact HCC-Embedded about their requirements.

The **K9F2816X0C.c** module is dependent on the specific flash device used and its configuration – i.e. which manufacturer, what size is the chip, is the data interface 8 or 16 bit and are there several chips in 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 ***fs_mountdrive*** to initialize the physical driver.

These functions are fully documented below.

These functions then require subroutine calls to fulfill their function. After these function definitions a description of all the routines used in this module is given. These routines are documented for 2 * K9F2816X0C Samsung chips in a parallel configuration. For any specific device the implementation may vary. The routines are documented to give guidance as to how to implement this module.

Physical Device Usage

The developer has to make some decisions about how to use their flash device. To use a flash device the developer must be aware that all devices are divided into a set of erasable blocks. It is only possible to write to an erased area and it is not possible to erase anything smaller than a block and thus some complex management software is used.

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

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

The sections below describe how to assign these.

Reserved blocks

The developer can reserve as many blocks from the physical device as required for private usage. This is done simply by omitting those blocks from the ***GetBlockAddr*** function.

If the developer wants to access reserved blocks using the ***GetBlockAddr*** function then this may also be done by selecting the physical block numbers to be used and ensuring they are not used by those specified in the descriptor and file system usage below.

e.g. If a particular physical device has 1024 erasable blocks and the user wants to reserve 256 blocks from the beginning for private usage they might set:

maxblock = 768	-	number of blocks for use by the file system
blockstart = 256	-	first file storage block

Thus if the user requests ***GetBloackAddr*** for blocks 0-255 they will get the address of a block in the physical device not used by the file system.

Note: The developer should take care accessing 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.

Descriptor Blocks

These blocks contain critical information about the file system, block allocation, 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:

descblocksize

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 separate blocks 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.

File System Blocks

The developer should allocate as many of these as required for their 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 these blocks that may be used by the file system. 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. This 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

Write Cache

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

To use the write cache the ***WriteVerifyPage*** function must be present. If this function does not exist then write caching will not be done.

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

`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.

Maximum Files

The maximum number of file/directory entries that can be made on a file system is restricted.

The maximum number of directory and file entries available on the system can 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.

Physical Layer Functions

fs_phy_nand_xxx

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

Format

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

Arguments

Argument	Description
flash	pointer to flash structure which 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 */
} FS_FLASH;
```

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 where to store data
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
any number	if there was any error during reading

Comments:

Blockrel is a number, which says the reading start position in block, could be a number from 0 to block size.

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

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

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 source data to be written
block	zero based block number where to store data
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

VerifyFlash

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

Format

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

Arguments

Argument	Description
data	points source data to be compared
block	zero based block number where to compare data
relsector	zero based relative sector in block
size	length of data need to 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 only required where this is the desired method of ensuring that the device has been correctly written. To decide whether to use a verify routine or not the device datasheet should be read. If, for example, ECC is being used and the reliability being guaranteed by this is sufficient for your requirements then the verify routine may be omitted. **This has a significant performance benefit.**

CheckBadBlock

This function is called at file system initialization to determine which blocks are bad blocks. The flash device may contain invalid blocks and in this function is called to sign them in for file system not to use. Higher level will call this function for all used block. The method how to check a block if it is bad 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

GetBlockSignature

This function is called from 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

WriteVerifyPage

This function verifies that a page of data within the flash matches a buffer containing the written data. This function 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 need 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 only required where this is the desired method of ensuring that the device has been correctly written. To decide whether to use a verify routine or not the device datasheet should be read. If, for example, ECC is being used and the reliability being guaranteed by this is sufficient for your requirements then the verify routine may be omitted. **This has a significant performance benefit.**

BlockCopy

This function copies one block to another block. This function is only called if static wear is being used. This routine should be implemented to use any features of the target device which may be used to accelerate a block to block copy operation. Many devices have features to support this which helps 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

Subroutine Descriptions and Notes for Sample Driver

This section contains a complete list of subroutines, describes their functionality and includes notes for porting these routines to a particular hardware design.

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 hi 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 temporally buffer (this buffer is 32bit 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 successfully, 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 need to copy and copy
ReadPage(pagenum) until datalength=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 if given block is bad or not
Calculate pagenum
Send read spare area command to NAND flash
Check 6^{th} word if its not 0xffffffff return with error, other case return 0 (ok)

GetBlockSignature(block: long)

Read signature data from block

fs_phy_nand_K9F2816X0C (flash: struct)

Set function pointers for driver.
Getting device ID and manufacture ID from NAND flash
Compare all supported device/manufacture and fills flash structure with corresponding data (size, sectors, block information)
If device not found returns with error

6 RAM Driver

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

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

2. After ***fs_init*** has been called, call the function ***fs_mountdrive*** with a pointer to the memory area you wish to use for the drive and the size of that area. e.g.

```
#define RAM_DRIVE_SIZE 0x1000000

void main(void){

      fs_init();              /* initialize the file system */

      /* mount first drive – A */

      fs_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.

7 File System Test

Supplied with the system is test code for exercising the system and ensuring that the file system is working correctly. 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.c** and **test.h** in your test project.

`void fs_dotest(void)` is called to execute the test code.

The test program requires the following three functions to be implemented by the developer - they are host system dependent - sample code below demonstrates the required functionality:

```
/* int _fs_poweron(void)            */
/* the developer should provide this function which should call */
/* f_initvolume for the drive to be tested - which must be drive 0 */
/* ("A"). If the RAM drive is being tested then the volume must be */
/* initialized and then formatted (f_initvolume then f_format). */
/* _f_poweron is called by the test code during the test operation. */
/* This routine should return non-zero if any error is detected. */

int _fs_poweron(void)
{
   /* A sample of this function is included in /serc/test/main.c */
}

/* _fs_dump() displays text from the running tests */

void _fs_dump (char *s)
{
      printf("%s\n",s);
}

/* _f_result() function to display errors detected during the test */

long _fs_result(long testnum, long error)
{
      printf("test number %d failed with error %d/n", testnum, error);
      return(testnum)
}
```