µC/FS

File System

Software Version 1.26

CPU independent

User & Reference manual

Micriμm Technologies Corporation

www.micrium.com
Empowering Embedded Systems

Disclaimer

The information in this document is subject to change without notice. While the information herein is assumed to be accurate, Micriµm Technologies Corporation (the vendor) assumes no responsibility for any errors or omissions.

The vendor makes and you receive no warranties or conditions, express, implied, statutory or in any communications with you. The vendor specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact

Micrium Technologies Corporation
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037

Email: support@micrium.com
Web: http://www.micrium.com/

Software and manual versions

This manual describes the software version 1.10. If any error occurs, please inform us and we will try to assist you as soon as possible.
For further information on topics or routines not yet specified, please contact us.

Print date: 1/13/2003

Software	Manual	Date	By	Explanation
1.26	1	030113	TB	Support for trial version added. FS__fat_malloc returns cleared buffer. MMC driver deactivates CS whenever allowed by specification.
1.24	1	021205	TB	FAT32 & POSIX like directory functions added
1.20	1	021010	TB	IDE & CompactFlash driver added
1.10	1	020927	TB	MultiMedia & SD card driver added
1.00	1	020830	KG	Revised for language/grammar. Version control table added. Typographic conventions (section 1.3) changed into table. Index added.

Contents

Disclaimer ... 2
Copyright notice ... 2
Trademarks ... 2
Contact ... 2
Software and manual versions ... 3
Contents ... 4
1. About this document ... 6
    1.1. Assumptions ... 6
    1.2. How to use this manual ... 6
    1.3. Typographic Conventions for Syntax ... 6
2. Introduction to *µC/FS* ... 7
    2.1. What is *µC/FS*? ... 7
    2.2. Features ... 7
3. Basic concepts ... 8
    3.1. API Layer ... 8
    3.2. File System Layer ... 8
    3.3. Logical Block Layer ... 9
    3.4. Device Driver ... 9
4. Getting started ... 10
    4.1. Installation ... 10
    4.2. Use the Windows sample ... 10
    4.3. Integrating *µC/FS* into your system ... 15
5. Configuration of *µC/FS* ... 17
    5.1. fs_conf.h ... 17
    5.2. fs_port.h ... 21
6. API functions ... 24
    6.1. File system control functions ... 25
    6.2. File access functions ... 27
    6.3. Direct input/output functions ... 30
    6.4. File positioning functions ... 32
    6.5. Error-handling functions ... 34
    6.6. Operations on files ... 36
    6.7. Directory functions ... 37
7. Device drivers ... 43
    7.1. Device driver functions ... 43
    7.2. Device driver function table ... 48
    7.3. Integrate a new device driver ... 48
8. SmartMedia Card Device Driver ... 50
    This chapter describes all hardware access functions required by *µC/FS*'s generic *SmartMedia* driver. ... 50
    8.1. Control line functions ... 51
    8.2. Power control functions ... 57
    8.3. Status detection functions ... 59
    8.4. Data transfer functions ... 66
    8.5. Timer functions ... 68
9. *MultiMedia & SD* card device driver ... 72
    9.1. Control line functions ... 72
    9.2. Operation condition detection and adjusting functions ... 76
    9.3. Status detection functions ... 79
    9.4. Data transfer functions ... 82
10. *CompactFlash* card & IDE device driver ... 88
    10.1. Control line functions ... 88

10.2. ATA I/O register access functions 91
10.3. Status detection functions 109
11. OS integration 111
11.1. OS layer control functions 111
11.2. Internal data structure protection 113
11.3. File access protection 117
11.4. Device access protection 119
11.5. Time/Date functions 121
Index 123

1. About this document

This guide describes the functionality and user API of *µC/FS* FAT File System.

1.1. Assumptions

This guide assumes that you already have a solid knowledge of the following:

- The software-tools used to build your application (assembler, linker, "C"-compiler)
- The C-language
- The target processor
- DOS-command-line

If you feel your knowledge of C is not good enough, we recommend *The C Programming Language* by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and in newer editions also covers ANSI C.

1.2. How to use this manual

This manual explains all the functions and macros that µC/FS offers. It assumes you have a working knowledge of the C-Language, knowledge of assembly programming is not required.

1.3. Typographic Conventions for Syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (i.e. system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
Emphasis	Very important sections.
Term	Important terms.

2. Introduction to *µC/FS*

2.1. What is *µC/FS*?

µC/FS is a FAT file system which can be used on any media, for which you can provide basic hardware access functions. .
µC/FS is a high performance library that has been optimized for speed, versatility and memory footprint.

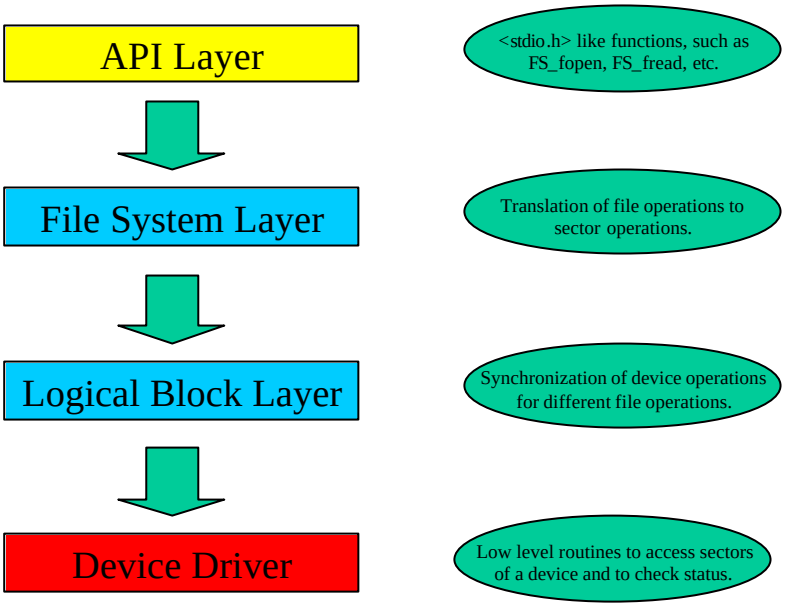
2.2. Features

µC/FS is written in ANSI C and can be used on virtually any CPU. Some features of *µC/FS* are:

- MS-DOS/MS-Windows compatible FAT12 and FAT16 support.
- Multiple device driver support. You can use different device drivers with *µC/FS,* which allows you to access different types of hardware with the file system at the same time.
- Multiple media support. A device driver does allow you to access different medias at the same time.
- OS support. *µC/FS* can easily be integrated into any OS. In that way you can make file operations in a multithreaded environment.
- ANSI C stdio.h like API for user applications. An application using standard C I/O library can easily be ported to use *µC/FS.*
- Very simple device driver structure. *µC/FS* device drivers need only very basic functions for reading and writing blocks. Therefore it is very simple to support your custom hardware.
- Generic device driver for *SmartMedia* cards, which can easily be used with any kind of card reader hardware.
- Generic device driver for *MultitMedia & SD* cards using SPI mode, which can be easily integrated.

3. Basic concepts

µC/FS is organized in different layers. In this chapter, you will find a short description of the functionality of each layer.



3.1. API Layer

The API layer is the interface between *µC/FS* and the user application. It does contain a library of ANSI C oriented file functions, such as `FS_FOpen`, `FS_FWrite` etc. The API layer does transfer these calls to the file system layer. Currently there is only a FAT file system layer available for *µC/FS*, but the API layer can deal with different file system layers at the same time. So it is possible to use FAT and any other file system at the same time with *µC/FS*.

3.2. File System Layer

The file system *layer* translates file operations to logical block operations. After such a translation, the file system calls the logical block layer and specifies the corresponding device driver for a device.

3.3. Logical Block Layer

The main purpose of the *logical block layer* is to synchronize accesses to a device driver and to have an easy interface for the file system layer. The logical block layer does call a device driver to make a block operation.

3.4. Device Driver

Device drivers are low level routines that are used to access your hardware. The structure of the device driver is simple to allow easy integration of your own hardware.

4. Getting started

This chapter gives you a step by step introduction on how to use *µC/FS*.

4.1. Installation

µC/FS is shipped on CD-ROM or as a `.zip` file in electronic form.

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, please keep all files in their respective sub-directories. Make sure the files are not read-only after copying.

If you received `.zip` file, please extract it to any folder of your choice, preserving the directory structure of the `.zip` file.

4.2. Use the Windows sample

If you have MS Visual C++ 6.00 or later version available, you will be able to work with a Windows sample project using *µC/FS*. Even if you do not have the Microsoft compiler, you should read this chapter to understand how an application can use *µC/FS*.

4.2.1. Building the sample program

Open the Workspace `I86_MS_Start.dsw` with MS Visual Studio by e.g. double clicking it. There is no further configuration necessary. You should be able to build the application without any error or warning message.

4.2.2. Stepping through the sample

The sample project uses the RAM disk driver for demonstration. It will create two files in the RAM disk and then dump contents of these files to the console window.

After starting the debugger by stepping into the application, your screen should look as shown in Figure 4-1:

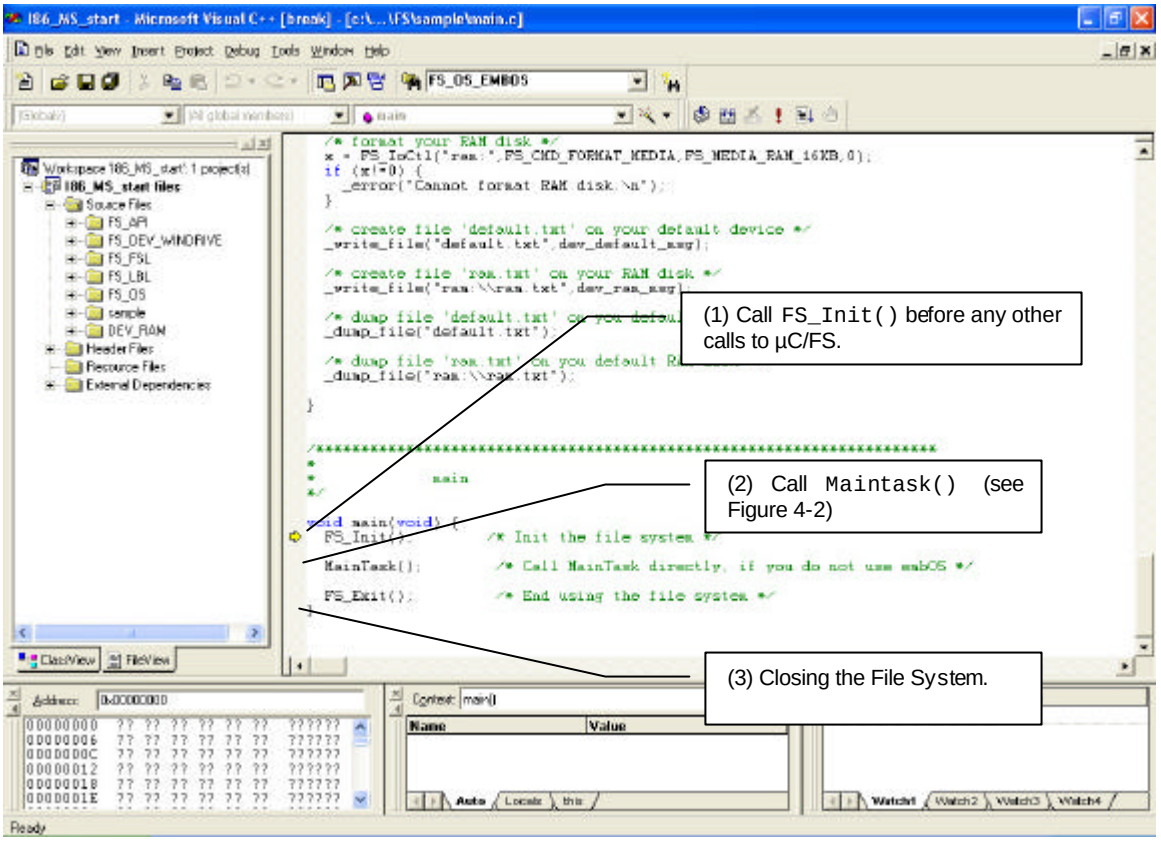


Figure 4-1

F4-1(1) One of the first things called in the sample program is a call to the *µC/FS* function `FS_Init()`. This function initializes the file system and must be called before using any other *µC/FS* function. You should step over (F10) this function.

F4-1(2) When you reach the call to `MainTask()`, you should step into (F11) this function and your screen should look as shown in Figure 4-2:

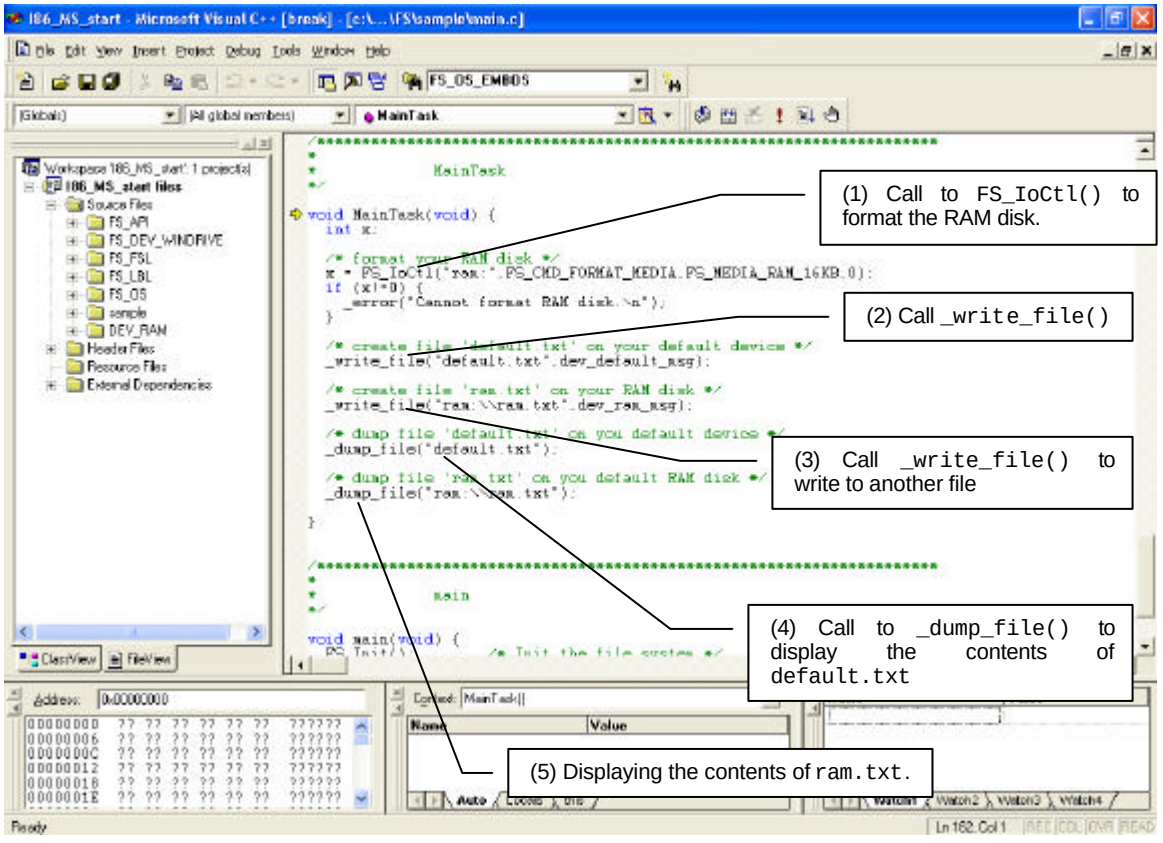


Figure 4-2

F4-2(1) There is a call to the µC/FS function `FS_IoCtl()`. This call is used to format your RAM disk in order to be able to write data to it. Formatting your RAM disk should not cause any problem.

F4-2(2) You should get to the call of function `_write_file()` without any error message. Please note that `_write_file()` is not a *µC/FS* function -- it is part of the sample application. After stepping into (F11) that function, your screen should look as shown in Figure 4-3:

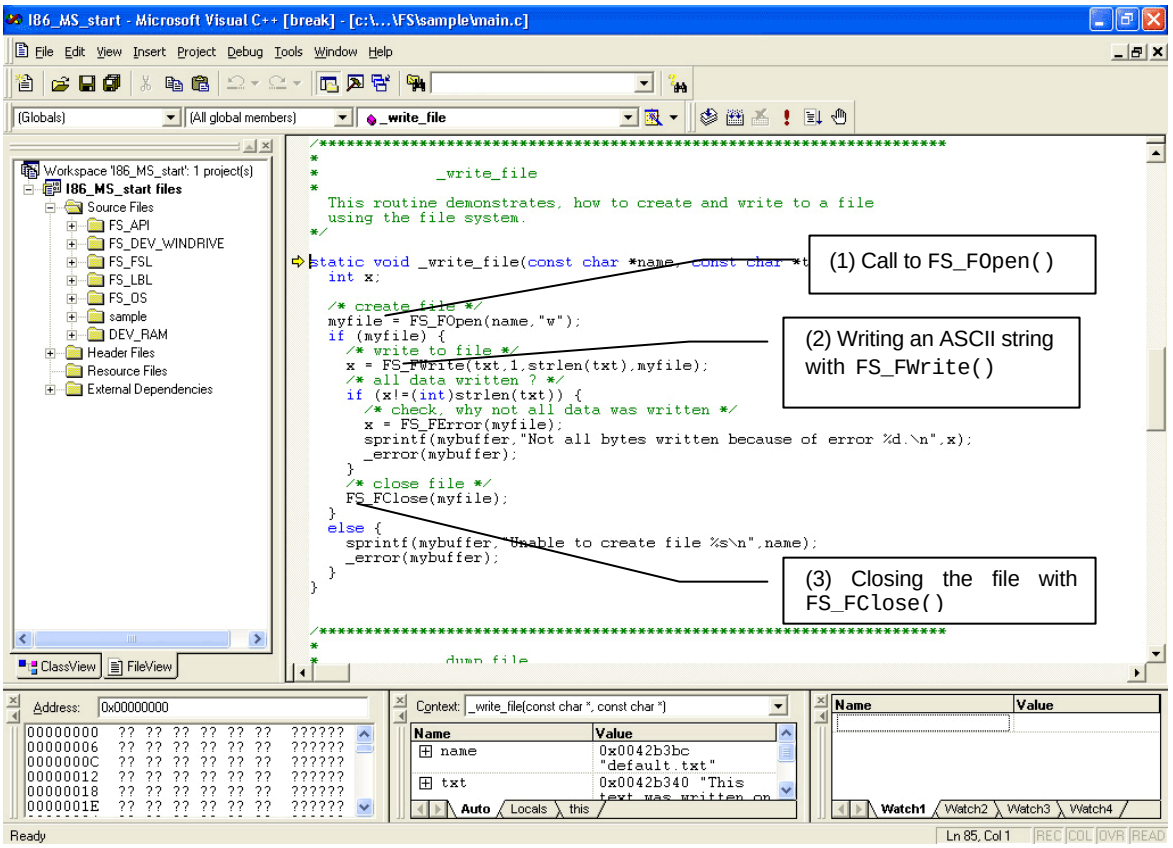


Figure 4-3

F4-3(1) You should get to the *µC/FS* function call `FS_FOpen()`. This function creates a file named `default.txt` in the root directory of your default device, which is the RAM disk in this example. Step over (F10) this function, which should return the address of an `FS_FILE` structure. In case of any error, it would return 0, indicating that the file could not be created.

F4-3(2) If `FS_FOpen` returns a valid pointer to an `FS_FILE` structure, the sample application will write a small ASCII string to this file by calling the µC/FS function `FS_FWrite()`. Step over (F10) this function and watch the return value. If no problem occurs, it should match with the length of the text. In case there have been less characters written, the sample project checks for an error code by calling the *µC/FS* function `FS_FError()`.

F4-3(3) Continue stepping over (F10) until you reach the call to *µC/FS* function `FS_FClose()`. This will close the file “default.txt”. Step over (F10) this function and continue stepping over (F10) until you are back to `MainTask()` (see Figure 4-2).

F4-2(3) You will get to the next call of `_write_file()`, which creates another file called “ram.txt” on your RAM disk. The difference between this and the first call is that this time, the RAM disk is specified explicitly. So the file will be created on your RAM disk, even if you use a different default device.

F4-2(4) Continue stepping over (F10) until you reach the call to `_dump_file()`. Note that this function is also part of the example and not *µC/FS*. After stepping into (F11) the function, your screen should look as shown in Figure 4-4:

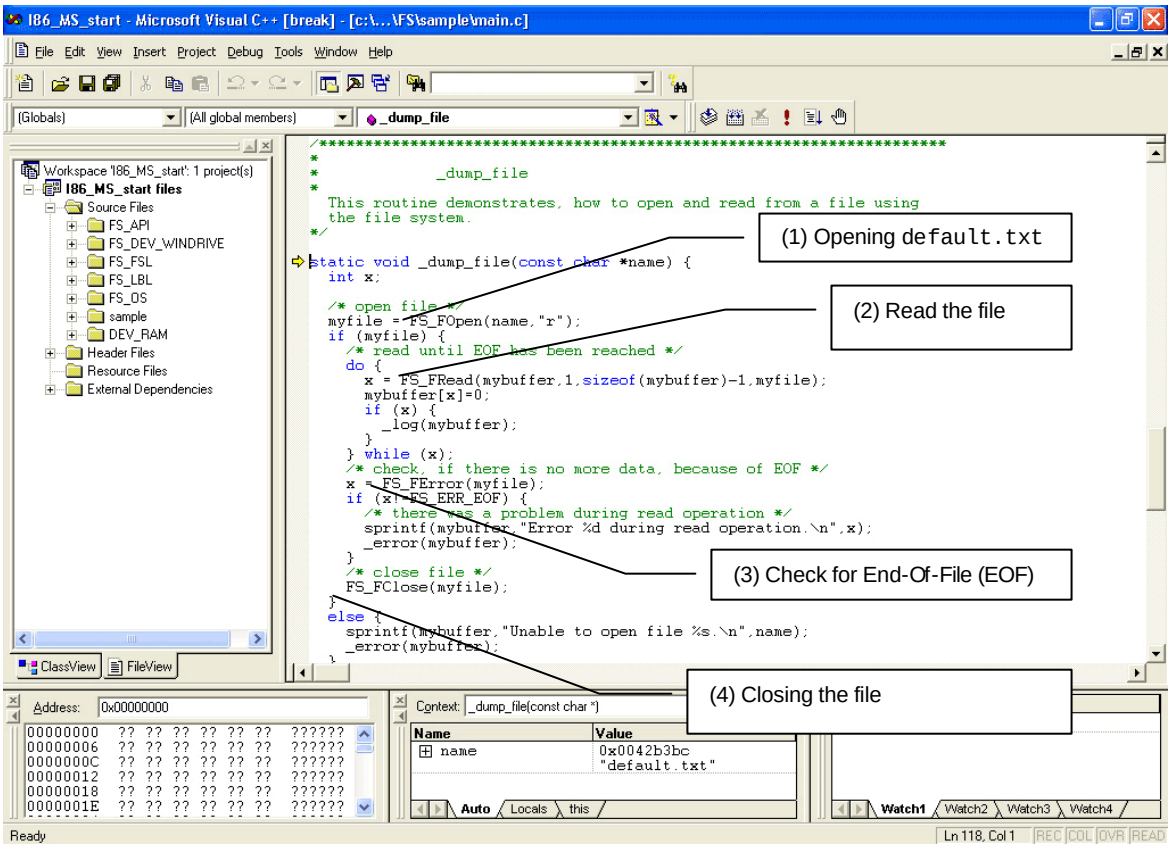


Figure 4-4

F4-4(1) The sample application opens the previously created file `default.txt` for read operation by using the *µC/FS* function `FS_FOpen()`. If the file does not exist or cannot be opened for any reason, `FS_FOpen()` will return 0 instead of a valid pointer to an `FS_FILE` structure.

F4-4(2) Continue stepping over (F10) until you get to the *µC/FS* function call `FS_FRead()`. Watch the contents of `mybuffer` and the return value when stepping over (F10) the function. You should see that `FS_FRead()` loads the contents of the file `default.txt` into `mybuffer` and returns the length of the string. If `FS_FRead()` returned data, it would be displayed on the console window with help of the sample application function `_log`.

F4-4(3) Continue stepping over (F10), until you get to the second call to `FS_FRead()`. This time, `FS_FRead()` returns 0, because the file pointer has reached the end of file `default.txt`. The sample application calls `FS_FError()` to see if EOF (i.e. the End-Of-File) has been reached.

F4-4(4) Finally, `default.txt` is closed by calling `FS_FClose()`. Continue stepping over (F10), until you get back to `MainTask()` (see Figure 4-2).

F4.2(5) The second call to `_dump_file()` in `MainTask()` does the same thing as the previous call except that it displays the contents of ram.txt, the second file we wrote. Continue stepping over (F10) until you get back to the function `main()`, in Figure 4-1.

F4-1(3) If your application terminates and thus, you don't need to use the file system, you can close the whole file system by calling `FS_Exit()`. This function call stops *µC/FS* and you MUST NOT call any *µC/FS* functions except `FS_Init()` which is used to re-initialize the file system.

Congratulations, you have successfully built and debugged your first application using *µC/FS*.

4.3. Integrating *µC/FS* into your system

In this chapter, you will learn how to add µC/FS to your system.

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You must thus be able to add files, directories to the include search path, etc. It is also assumed, that you are familiar with the OS you will be using in your target system, if you are using one.

4.3.1. Create a simple project without *µC/FS*

We recommend, that you create a small “hello world” program for your system. That project should already use your OS and there should be a way to display text on a screen or serial port.

4.3.2. Add your *µC/FS* configuration

In order to configure *µC/FS* for your system, you should create a new sub-directory in µC/FS’s config directory and copy the files `fs_conf.h` and `fs_port.h` from one of the other sub-directories to your directory. For the following chapters, we assume that you have created a directory `FS\CONFIG\myconfig`. Usually, the only file you have to modify is `fs_conf.h`. For an easy startup, we recommend, that you disable all drivers except the RAM disk driver. Please check out the chapter “Configuration of *µC/FS*” for detailed information about the configuration.

4.3.3. Add *µC/FS* generic source code

Add all source files in the following directories:

```
FS\API
FS\FSL
FS\LBL
FS\OS
FS\DEVICE\RAM
```

and their sub-directories to your project.

4.3.4. Configure the search path

In order to build all the files that you added, you will have to add the following directories to your path for include files:

```
FS\API
FS\CONFIG\myconfig
FS\LBL
FS\OS
```

4.3.5. Add generic sample code

For a quick and easy test of your *µC/FS* integration, you should use the code found in `FS\sample\main.c`.

4.3.6. Build and test your application

If everything is configured correctly, you should now be able to build an application containing a RAM disk on your target system.. If you encounter any problem during the build process, please check your include path and your

configuration. Usually *µC/FS*'s source code builds without any compiler warning.

Once you can build the application, you should be able to use it with your target's debugger environment just like we did with the Visual C++ sample project.

Congratulations, you have successfully integrated *µC/FS* into your system. .

5. Configuration of *µC/FS*

In your configuration directory for *µC/FS*, you will find two files, which allow you to configure the file system according to your requirements. In this chapter, you will find a description of each of these files.

5.1.fs_conf.h

This is the main configuration file for the file system. You define which drivers you want to use and, the configurations for these drivers.

Example

```
#ifndef _FS_CONF_H_
#define _FS_CONF_H_

/*
*********************************************************************
*                          Micrium, Inc.
*                      949 Crestview Circle
*                     Weston,  FL 33327-1848
*
*                            uC/FS
*
*                (c) Copyright 2002, Micrium, Inc.
*                      All rights reserved.
*
*********************************************************************

---------------------------------------------------------------------
File        : fs_conf.h
Purpose     : File system configuration
              Usually all configuration of the file system for your
              system can be done by changing this file.
               If you are using a big endian system or a totally
              different architecture, you may have to modify the file
              "fs_port.h".
---------------------------------------------------------------------
Version-Date-----Author-Explanation
---------------------------------------------------------------------
1.00.00 20020815        First release
---------------------------------------------------------------------
Known problems or limitations with current version
---------------------------------------------------------------------
None.
---------------------------END-OF-HEADER-----------------------------
*/

/*********************************************************************
*
*             #define constants
*
*********************************************************************
*/


/*********************************************************************
*
*             Number of file handles
*
    Set the maximum number of simultaneously open files in your system.
    Please be aware, that the file system requires one FS_FILE structure
    for each open file. If you are using FAT, each file will also require
    two sector buffers.
*/

#define FS_MAXOPEN            4    /* Maximum number of file handles */
```

```
/*********************************************************************
*
*             OS Layer
*
    Set all to 0, if you do not want OS support.
*/

#define FS_OS_EMBOS             1    /* 1 = use embOS */
#define FS_OS_UCOS_II           0    /* 1 = use uC/OS-II */
#define FS_OS_WINDOWS           0    /* 1 = use WINDOWS */
#if ((FS_OS_WINDOWS==1) && ((FS_OS_EMBOS==1) || (FS_OS_UCOS_II==1)))
  #error You must not use Windows at the same time as embOS or uC/OS-II!
#endif


/*********************************************************************
*
*             Time/Date support
*
  If your system does support ANSI C library functions for time/date,
  you can set this to 1. If it is set to 0, functions FS_OS_Get_Date
  and FS_OS_Get_Time will return the date 1.01.1980 0:00 unless you
  modify them.
*/

#define FS_OS_TIME_SUPPORT      0    /* 1 = time()/date supported */


/*********************************************************************
*
*             File System Layer
*
  You can turn on/off the file system layers used in your system.
  At least one layer has to be active. Because currently there is only
  one file system layer supported (FAT), you will usually not change
  this setting.
*/

#define FS_USE_FAT_FSL          1    /* FAT12/FAT16 file system */


/*********************************************************************
*
*             Device Driver Support
*
  You can turn on/off device driver supported in your system.
  If you turn on a driver, please check also its settings in this
  file below.
*/

#define FS_USE_SMC_DRIVER       1    /* SmartMedia card driver       */
#define FS_USE_WINDRIVE_DRIVER  0    /* Windows Logical Drive driver */
#define FS_USE_RAMDISK_DRIVER   1    /* RAM Disk driver              */

#if (!defined(_WIN32) && (FS_USE_WINDRIVE_DRIVER))
  #error Windows Logical Drive driver needs Windows API
#endif

/*********************************************************************
*
*             FAT File System Layer defines
*/

#if FS_USE_FAT_FSL
  /*
     For each media in your system using FAT, the file system reserves
     memory to keep required information of the boot sector of that media.
     FS_MAXDEV is the number of device drivers in your system used
     with FAT, FS_FAT_MAXUNIT is the maximum number of logical units
     supported by one of the activated drivers.
  */
  #define FS_MAXDEV             (FS_USE_SMC_DRIVER + FS_USE_WINDRIVE_DRIVER +
FS_USE_RAMDISK_DRIVER)
```

```
  #define FS_FAT_MAXUNIT        2         /* max number of medias per device
*/

  /*
    Do not change following defines !
    They may become configurable in future versions of the file system.
  */
  #define FS_FAT_SEC_SIZE       0x200     /* do not change for FAT */
  #define FS_FAT_MAXCLST        100000    /* max cluster number (do not
change) */
#endif /* FS_USE_FAT_FSL */


/*********************************************************************
*
*             RAMDISK_DRIVER defines
*/

#if FS_USE_RAMDISK_DRIVER
  /*
    Define size of your RAM disk here.
    You specify the number of sectors (512 bytes) here.
  */
  #define FS_RR_BLOCKNUM      32        /* 16KB RAM */
  /*
    Do not change following define !
    It may become configurable in future versions of the file system.
  */
  #define FS_RR_BLOCKSIZE     0x200     /* do not change for FAT */
#endif  /* FS_USE_RAMDISK_DRIVER */


/*********************************************************************
*
*             SMC_DRIVER defines
*
  Settings of the generic Smartmedia Card driver.
  For using SMC in your system, you will have to provide basic
  hardware access functions. Please check device\smc\hardware\XXX\smc_hw.c
  and device\smc\hardware\XXX\smc_hw.h for samples.
*/

#if FS_USE_SMC_DRIVER
  /*
    Number of card readers in your system.
    Please note, that even if your system does have more than one
    SMC slot, it might be unable to access them both at the same
    time, because they share resources (e.g. same data port). If that
    is true for your system, you must ensure, that your implementation
    of "FS_OS_Lock_deviceop(&_FS_smcdevice_driver,id)" blocks the device
    driver for all values of "id"; the default implementation of the
    file system's OS Layer does so.
  */
  #define FS_SMC_MAXUNIT                    2  /* EP7312 SMC + on board NAND
*/
  /*
    The following define does tell the generic driver, if your
    system can directly check the SMC RY/BY line.
  */
  #define FS_SMC_HW_SUPPORT_BSYLINE_CHECK   0  /* EP7312 does not support */
  /*
    FS_SMC_HW_NEEDS_POLL has to be set, if your SMC hardware driver
    needs to be called periodically for status check (e.g. diskchange).
    In such a case, the generic driver does provide a function
    "void FS_smc_check_card(FS_u32 id)", which has to be called
    periodically by your system for each card reader.
  */
  #define FS_SMC_HW_NEEDS_POLL              1  /* EP7312 needs poll for
diskchange */
#endif /* FS_USE_SMC_DRIVER */


/*********************************************************************
*
*             WINDRIVE_DRIVER defines
```

```
*
  This driver does allow to use any Windows logical driver on a
  Windows NT system with the file system. Please be aware, that Win9X
  is not supported, because it cannot access logical drives with
  "CreateFile".
*/

#if FS_USE_WINDRIVE_DRIVER
  /*
    The following define tells WINDRIVE, how many logical drives
    of your NT system you are going to access with the file system.
    if your are going to use more than 2 logical drives, you
    will have to modify function "_FS_wd_devstatus" of module
    device\windriver\wd_misc.c.
  */
  #define FS_WD_MAXUNIT         2             /* number of windows drives */
  /*
    Specify names of logical Windows drives used with the file system. For
example,
    "\\\\.\\A:" is usually the floppy of your computer.
  */
  #define FS_WD_DEV0NAME        "\\\\.\\E:"   /* Windows drive name for
"windrv:0:" */
  #define FS_WD_DEV1NAME        "\\\\.\\A:"   /* Windows drive name for
"windrv:1:" */
  /*
    To improve performance of WINDRIVE, it does use sector caches
    for read/write operations to avoid real device operations for
    each read/write access. The number of caches can be specified
    below and must not be smaller than 1.
  */
  #define FS_WD_CACHENUM        40            /* number of read caches per
drive */
  #define FS_WD_WBUFFNUM        20            /* number of write caches per
drive */
  /*
    Do not change following define !
    It may become configurable in future versions of the file system.
  */
  #define FS_WD_BLOCKSIZE       0x200         /* do not change for FAT */
#endif  /* FS_USE_WINDRIVE_DRIVER */


#endif  /* _FS_CONF_H_ */
```

5.1.1.OS support

You can specify whether you are using uC/OS-II, embOS, Windows or no OS support at all. Please set `FS_OS_UCOS_II`, `FS_OS_EMBOS`, `FS_OS_WINDOWS` to 1, respectively. For no OS support at all, set all of them to 0. If you need support for an additional OS, you will have to provide functions described in the chapter “OS integration”.

5.1.2.Time/Date support

If you want to be able to add date and times to your files, you will need to set `FS_OS_TIME_SUPPORT` to 1.

5.1.3.File System Layer Support

µC/FS can support different file system at the same time. You can enable them by setting `FS_USE_XXX_FSL`, where XXX is the name of the file system layer. The current version of µC/FS only supports the FAT file system, so you will need to set `FS_USE_FAT_FSL` to 1.

5.1.4.Device Driver Support and configuration

µC/FS is shipped with three device drivers:

RAM Disk: *µC/FS* allows you to create any number of RAM disk drives. To support these types of drives, simply set `FS_USE_RAMDISK_DRIVER` to 1. By default, *µC/FS* implements a RAM disks with 16Kbytes of storage.

Windows Driver: This driver allows you to use Microsoft Windows as the vehicle to store your files. However, this driver only works with Windows NT or Windows 2000. This driver allows you to access real Windows drives (e.g. floppy or flashcards) with the help of the Windows API. It is very useful for testing. This driver is enabled by setting `FS_USE_WINDRIVE_DRIVER` to 1. note Please be careful when using this driver because it can destroy data on your hard disk.

***Smart Media* Card**: *µC/FS* can work with Smart Media Cards also known as SMCs. You can enable it by setting `FS_USE_SMC_DRIVER` to 1. In order to use it, you will have to provide low level I/O functions for your card reader hardware. Please check out chapter "*SmartMedia card device driver*" for details.

***MultiMedia* card**: *µC/FS* can support *MultiMedia & SD* cards. You can enable the driver by setting `FS_USE_MMC_DRIVER` to 1. In order to use it, you will have to provide low-level I/O functions for your card reader hardware. Please take a look at the chapter "*MultiMedia & SD* card device driver" for details.

***CompactFlash* card & IDE**: *µC/FS* supports *CompactFlash* storage cards, also known as CF. The driver uses *true IDE* mode to access a card. You can use the same driver for accessing ATA HD drives. To enable the driver, set `FS_USE_IDE_DRIVER` to 1. In order to use it, you will have to provide low-level I/O functions for your card reader or IDE port hardware. Please take a look at the chapter "*CompactFlash* card & IDE device driver" for details.

5.2.fs_port.h

Usually this file only requires minor modifications, if you are using a very specific CPU. Please also check the type declarations in this file to ensure that they fit with your target processor and compiler.

Example

```
#ifndef _FS_PORT_H_
#define _FS_PORT_H_

/*
*********************************************************************
*                          Micrium, Inc.
*                      949 Crestview Circle
*                     Weston,  FL 33327-1848
*
*                            uC/FS
```

```
*
*                 (c) Copyright 2002, Micrium, Inc.
*                       All rights reserved.
*
*********************************************************************
---------------------------------------------------------------------
File        : fs_port.h
Purpose     : Architecture dependend defines for the file system
              This header defines basic data types and access macros
              used by the file system. Depending on your CPU/compiler,
              you may have to modify this file.
---------------------------------------------------------------------
Version-Date-----Author-Explanation
---------------------------------------------------------------------
1.00.00 20020815        First release
---------------------------------------------------------------------
Known problems or limitations with current version
---------------------------------------------------------------------
None.
---------------------------END-OF-HEADER-----------------------------
*/

/*********************************************************************
*
*             #define constants
*
*********************************************************************
*/

#define FS_LITTLE_ENDIAN      1     /* 1 = little endian, 0 = big endian */


/*********************************************************************
*
*             define global data types
*
*********************************************************************

  These defines will work for most 8/16/32 bit CPUs. Please check
  your compiler manual if you are not sure.
*/

#define FS_size_t unsigned long       /* 32 bit unsigned */
#define FS_u32    unsigned long       /* 32 bit unsigned */
#define FS_i32    signed long         /* 32 bit signed */
#define FS_u16    unsigned short      /* 16 bit unsigned */
#define FS_i16    signed short        /* 16 bit signed */


/*********************************************************************
*
*             #define access macros
*
*********************************************************************

  Following macros are used to access data of a FAT file system.
  Systems using little endian can access data as it is, but big
  endian systems have to swap.
*/

#if (FS_LITTLE_ENDIAN==1)
#define FS__r_u32(value)   value
#define FS__w_u32(value)   value
#define FS__r_u16(value)   value
#define FS__w_u16(value)   value
#else
#define FS__r_u32(value)   ((value >> 24) | (value << 24) | ((value &
0x00ff0000ul) >> 8) | ((value & 0x0000ff00ul) << 8))
#define FS__w_u32(value)   _FS_r_u32(value)
#define FS__r_u16(value)   ( ((u_short)(value>>8) | (u_short)(value<<8)) )
#define FS__w_u16(value)   _FS_r_u16(value)
#endif
#endif  /* _FS_PORT_H_ */
```

6. API functions

In this chapter, you will find a description of each *µC/FS* API functions. An application should only access *µC/FS* by these functions. The table below lists the available API functions within their respective categories.

Routine	Explanation
File system control functions	
FS_Exit()	Stop file system
FS_Init()	Start file system
File access functions	
FS_FClose()	Close a file
FS_FOpen()	Open a File
Direct input/output functions	
FS_FRead()	Read data from file
FS_FWrite()	Write data to file
File positioning functions	
FS_FSeek()	Set position of file pointer
FS_FTell()	Return position of file pointer
Error-handling functions	
FS_ClearErr()	Clear error status
FS_FError()	Return error code
Operations on files	
FS_Remove()	Delete a file

6.1. File system control functions

6.1.1. FS_Exit() : Stop file system

Description

Stop the file system

Prototype

```
void  FS_Exit(void);
```

Return value

void

Additional information

`FS_Exit()` removes e.g. resources required for an OS integration of *µC/FS* and stops the file system. After calling this function, you must not call any other *µC/FS* function except FS_Init().

Example

See `FS_Init()`.

6.1.2. FS_Init() : Start file system

Description

Start the file system

Prototype

```
void  FS_Init(void);
```

Return value

void

Additional information

`FS_Init()` initializes the file system and creates resources required for an OS integration of *µC/FS*. This function has to be called before calling any other *µC/FS* function.

Example

```
#include “fs_api.h”

void main(void) {
  FS_Init();
  /*
     access file system
  */
  FS_Exit();
}
```

6.2. File access functions

6.2.1. FS_FClose() : Close a file

Description

Close an open file

Prototype

```
void FS_FClose(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE

Return value

void

Additional information

Close an open file.

Example

```
void foo(void) {
  FS_FILE *myfile;

  myfile = FS_FOpen(“test.txt”,”r”);
  if (myfile!=0) {
    /*
       access file
    */
    FS_FClose(myfile);
  }
}
```

6.2.2. FS_FOpen() : Open a file.

Description

Opens an existing file or creates a new file depending on parameters.

Prototype

```
FS_FILE *FS_FOpen(const char *name, const char *mode);
```

Parameter	Meaning
name	Fully qualified file name.
mode	Mode for opening the file.

Return value

It returns the address of a `FS_FILE` structure, if the file could be opened in the requested mode. In case of any error the return value is 0.

Additional information

A fully qualified file name means:

```
[DevName:[UnitNum:]][DirPathList]FileName
```

`DevName` is the name of a device, e.g. “smc”, “ram” or “windrv”. If not specified, the first device in the device table will be used.

`UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify `UnitNum`, if `DevName` has not been specified.

`DirPathList` means a complete path to an already existing subdirectory; `FS_Fopen()` does not create directories. The path must start and end with a ‘\’ character. Directory names in the path are separated by ‘\’. If `DirPathList` is not specified, the root directory on the device will be used.

`FileName` and all directory names have to follow the standard FAT naming conventions (i.e. 8.3 notation); long file names are not supported.

The parameter `mode` points to a string. If the string is one of the following, *µC/FS* will open the file in the indicated mode:

Mode	Meaning
r	Open text file for reading
w	Truncate to zero length or create text file for writing
a	Append; open or create text file for writing at end-of-file
rb	Open binary file for reading
wb	Truncate to zero length or create binary file for writing
ab	Append; open or create binary file for writing at end-of-file
r+	Open text file for update (reading and writing)
w+	Truncate to zero length or create text file for update
a+	Append; open or create text file for update, writing at end-of-file
r+b or **rb+**	Open binary file for update (reading and writing)
w+b or **wb+**	Truncate to zero length or create binary file for update
a+b or **ab+**	Append; open or create bin. file for update, writing at end-of-file

For more details on the `FS_FOpen()` function, please also refer to the ANSI C documentation regarding the `fopen()` function.

Note that *µC/FS* does not distinguish between binary and text mode, files are always accessed in binary mode.

Example

```
FS_FILE *myfile;

void foo1(void) {
  /* open file for reading – default driver on default device */
  myfile = FS_FOpen(“test.txt”,”r”);
}

void foo2(void) {
  /* create new file for writing  - default driver on default device */
  myfile = FS_FOpen(“test.txt”,”w”);
}

void foo3(void) {
  /* open file for reading in folder ‘mysub’
     - default driver on default device */
  myfile = FS_FOpen(“\\mysub\\test.txt”,”r”);
}

void foo4(void) {
  /* open file for reading - RAM device driver on default device */
  myfile = FS_FOpen(“ram:test.txt”,”r”);
}

void foo5(void) {
  /* open file for reading - RAM device driver on device number 2*/
  myfile = FS_FOpen(“ram:1:test.txt”,”r”);
}
```

6.3. Direct input/output functions

6.3.1. FS_FRead() : Read data from file

Description

Reads data from an open file.

Prototype

```
FS_size_t  FS_FRead(void *ptr, FS_size_t size,
                    FS_size_t n, FS_FILE *fp);
```

Parameter	Meaning
ptr	Pointer to a data buffer for storing data transferred from file.
size	Size of an element to be transferred from file to the data buffer
n	Number of elements to be transferred
fp	Pointer to a data structure of type FS_FILE

Return value

Number of elements transferred.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the `FS_FError()` function.

Example

```
Char buffer[100];

void foo(void) {
  FS_FILE *myfile;

  myfile = FS_FOpen(“test.txt”,”r”);
  if (myfile!=0) {
    do {
      i = FS_FRead(buffer,1,sizeof(buffer)-1,myfile);
      buffer[i]=0;
      if (i) {
        printf("%s",buffer);
      }
    } while (i);
    FS_FClose(myfile);
  }
}
```

6.3.2. FS_FWrite() : Write data to a file

Description

Writes data to an open file.

Prototype

```
FS_size_t  FS_FWrite(void *ptr, FS_size_t size,
                     FS_size_t n, FS_FILE *fp);
```

Parameter	Meaning
ptr	Pointer to data, which should be written to the file.
size	Size of an element to be transferred from buffer to the file.
n	Number of elements to be transferred to the file.
fp	Pointer to a data structure of type FS_FILE

Return value

Number of elements written.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
const char welcome[]="hello world\n";

void foo(void) {
  FS_FILE *myfile;

  myfile = FS_FOpen(“test.txt”,”w”);
  if (myfile!=0) {
    FS_FWrite(welcome,1,strlen(welcome),myfile);
    FS_FClose(myfile);
  }
}
```

..

6.4. File positioning functions

6.4.1. FS_FSeek() : Set position of file pointer

Description

Sets current position of the file pointer.

Prototype

```
Int FS_FSeek(FS_FILE *fp, FS_i32 offset, int whence);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.
offset	Offset for setting the file pointer position
whence	Mode for positioning file pointer

Return value

If file pointer has been positioned according to the parameters, the return value is 0. In case of an error, the return value is -1.

Additional information

Valid values for parameter whence are FS_SEEK_SET, FS_SEEK_CUR and FS_SEEK_END. For details on these parameters, please check ANSI C documentation about the fseek() function. Note that *µC/FS* does not currently support positioning the file pointer after end of a file.

Example

```
const char welcome[]="some text will be overwritten\n";

void foo(void) {
  FS_FILE *myfile;

  myfile = FS_FOpen("test.txt","w");
  if (myfile!=0) {
    FS_FWrite(welcome,1,strlen(welcome),myfile);
    FS_FSeek(myfile,-4,FS_SEEK_CUR);
    FS_FWrite(welcome,1,strlen(welcome),myfile);
    FS_FClose(myfile);
  }
}
```

6.4.2. FS_FTell() : Return position of file pointer

Description

Returns current position of the file pointer.

Prototype

```
FS_i32     FS_FTell(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.

Return value

Current position of file pointer in the file. On failure the return value is -1.

Additional information

Currently this function does simply return the file pointer element of the file's FS_FILE structure. Nevertheless you should not access the FS_FILE structure by yourself, because that data structure might change in future.

Example

```
const char welcome[]="hello world\n";

void foo(void) {
  FS_FILE *myfile;
  FS_i32 pos;

  myfile = FS_FOpen("test.txt","w");
  if (myfile!=0) {
    FS_FWrite(welcome,1,strlen(welcome),myfile);
    Pos = FS_FTell(myfile);
    FS_FClose(myfile);
  }
}
```

6.5. Error-handling functions

6.5.1. FS_ClearErr() : Clear error status

Description

Clears the error status of a file

Prototype

```
void       FS_ClearErr(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.

Return value

void

Additional information

Call this routine after you have detected an error so you can check for success of the next file operations.

Example

```
void foo(void) {
  FS_FILE *myfile;
  FS_i16 err;

  myfile = FS_FOpen(“test.txt”,”r”);
  if (myfile!=0) {
    err = FS_FError(myfile);
    if (err!=FS_ERR_OK) {
      FS_ClearErr(myfile);
    }
    FS_FClose(myfile);
  }
}
```

6.5.2. FS_FError() : Return error code

Description

Returns current error status of a file

Prototype

```
FS_i16     FS_FError(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.

Return value

Returns value not equal to FS_ERR_OK in case a file operation caused an error.

Additional information

The return value is not FS_ERR_OK only in case a file operation caused an error and the error was not cleared by calling FS_ClearErr() or any other operation that clears the previous error status..

The following error codes have specific meanings:

Mode	Meaning
FS_ERR_OK	No error
FS_ERR_EOF	End-of-file has been reached.
FS_ERR_DISKFULL	Unable to write data, because no more space on media.
FS_ERR_INVALIDPAR	A µ*C/FS* function has been called with an illegal parameter.
FS_ERR_WRITEONLY	A read operation has been made on a file open for writing only.
FS_ERR_READONLY	A write operation has been made on a file open for reading only.
FS_ERR_READERROR	An error occurred during read operation.
FS_ERR_WRITEERROR	An error occurred during write operation.
FS_ERR_DISKCHANGED	Media has been changed, although the file was still open.
FS_ERR_CLOSE	An error occurred during close operation.

Example

```
void foo(void) {
  FS_FILE *myfile;
  FS_i16 err;

  myfile = FS_FOpen("test.txt","r");
  if (myfile!=0) {
    err = FS_FError(myfile);
    FS_FClose(myfile);
  }
}
```

6.6. Operations on files

6.6.1. FS_Remove() : Delete a file

Description

Removes file from the media.

Prototype

```
int        FS_Remove(const char * name);
```

Parameter	Meaning
name	Full qualified file name.

Return value

`0` if the file has been removed. In case of an error, the return value is `-1`.

Additional information

Values for name are the same as for `FS_FOpen()`. Please refer `FS_FOpen()` for examples of valid names.

Example

```
void foo(void) {
  FS_Remove("test.txt");
}
```

6.7. Directory functions

6.7.1. FS_CloseDir(): Close a directory

Description

This functions closes a directory referred by parameter `dirp`

Prototype

```
int FS_CloseDir(FS_DIR *dirp);
```

Parameter	Meaning
`dirp`	Pointer to directory structure

Return value

0 if the directory has been closed.
In case of any error the return value is -1.

Example

```
void foo(void) {
  FS_DIR *dirp;
  struct FS_DIRENT *direntp;

  dirp = FS_OpenDir("");  /* Open root directory of the default device */
  if (dirp) {
    do {
      direntp = FS_ReadDir(dirp);
      if (direntp) {
        sprintf(mybuffer,"%s\n",direntp->d_name);
        _log(mybuffer);
      }
    } while (direntp);
    FS_CloseDir(dirp);
  }
  else {
    _error("Unable to open directory\n");
  }
}
```

6.7.2. FS_MkDir(): Create a directory

Description

Creates a new directory.

Prototype

```
int FS_MkDir(const char *dirname);
```

Parameter	Meaning
Dirname	Fully qualified directory name.

Return value

Returns 0, if the directory has been successfully created.
In case of any error the return value is -1.

Additional information

A fully qualified directory name has the same meaning as for function FS_OpenDir. Please note, that FS_MkDir will not create the whole DirPathList, it will only create a directory in an already existing path.

Example

```
void foo1(void) {
  /* create mydir in directory test – default driver on default device */
  err = FS_MkDir("\\test\\mydir");
}

void foo2(void) {
  /* create directory mydir – RAM device driver on default device */
  err = FS_MkDir("ram:\\mydir");
}
```

6.7.3. FS_OpenDir(): Open a directory

Description

Opens an existing directory for reading.

Prototype

```
FS_DIR *FS_OpenDir(const char *dirname);
```

Parameter	Meaning
`Dirname`	Fully qualified directory name.

Return value

Returns the address of an `FS_DIR` data structure if the directory could be opened.
In case of any error the return value is 0.

Additional information

A fully qualified directory name means:

```
[DevName:[UnitNum:]][DirPathList]DirectoryName
```

- `DevName` is the name of a device, e.g. “smc”, “ram” or “windrv”. If not specified, the first device in the device table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify `UnitNum` if `DevName` has not been specified.
- `DirPathList` means a complete path to an existing subdirectory.. The path must start and end with a ‘\’ character. Directory names in the path are separated by ‘\’. If `DirPathList` is not specified, the root directory on the device will be used.
- `DirectoryName` and all directory names have to follow the standard FAT naming conventions (e.g. 8.3 notation); long file names are not supported.

To open the root directory, simply use an empty string for `DirectoryName`.

Example

```
FS_DIR *dirp;

void foo1(void) {
  /* open directory test – default driver on default device */
  dirp = FS_OpenDir(“test”);
}

void foo2(void) {
  /* open root directory – RAM device driver on default device */
  dirp = FS_OpenDir(“ram:”);
}
```

6.7.4. FS_ReadDir(): Read from a directory

Description

Reads next directory entry in directory specified by `dirp`.

Prototype

```
struct FS_DIRENT *FS_ReadDir(FS_DIR *dirp);
```

Parameter	Meaning
`dirp`	Pointer to an opened directory

Return value

Returns a pointer to a directory entry.
If there is no more entry in the directory or in case of any error, 0 is returned.

Example

Please refer to `FS_CloseDir`.

6.7.5. FS_RewindDir(): Reset position in directory stream

Description

This function sets pointer for reading the next directory entry to the first entry in the directory.

Prototype

```
void FS_RewindDir(FS_DIR *dirp);
```

Parameter	Meaning
dirp	Pointer to directory structure

Return value

Void.

Example

```
void foo(void) {
  FS_DIR *dirp;
  struct FS_DIRENT *direntp;

  dirp = FS_OpenDir("ram:");  /* Open root directory of the RAM device */
  if (dirp) {
    /* display directory */
    do {
      direntp = FS_ReadDir(dirp);
      if (direntp) {
        sprintf(mybuffer,"%s\n",direntp->d_name);
        _log(mybuffer);
      }
    } while (direntp);
    /* rewind to 1st entry */
    FS_RewindDir(dirp);
    /* display directory  again */
    do {
      direntp = FS_ReadDir(dirp);
      if (direntp) {
        sprintf(mybuffer,"%s\n",direntp->d_name);
        _log(mybuffer);
      }
    } while (direntp);
    FS_CloseDir(dirp);
  }
  else {
    _error("Unable to open directory\n");
  }

}
```

6.7.6. FS_RmDir(): Remove a directory

Description

Deletes a directory

Prototype

```
int FS_RmDir(const char *dirname);
```

Parameter	Meaning
dirname	Fully qualified directory name.

Return value

Returns 0, if the directory has been successfully removed.
In case of any error the return value is -1.

Additional information

A fully qualified directory name has the same meaning as for function FS_OpenDir. FS_RmDir will not delete a directory, which is not empty.

Example

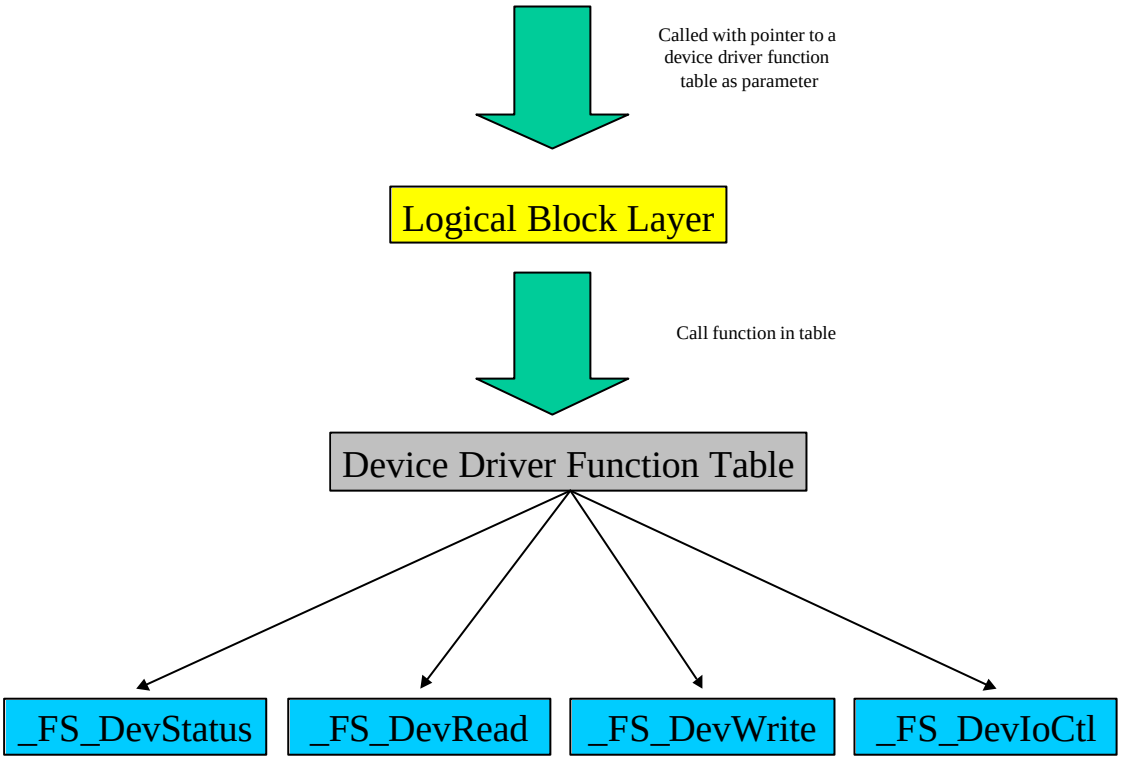
```
void foo1(void) {
  /* remove mydir in directory test – default driver on default device */
  err = FS_RmDir("\\test\\mydir");
}

void foo2(void) {
  /* remove directory mydir – RAM device driver on default device */
  err = FS_RmDir("ram:\\mydir");
}
```

7. Device drivers

µC/FS has been designed to cooperate with any kind of hardware. To use a specific hardware with *µC/FS*, a so called device driver for that hardware is required. The device driver consists of basic I/O functions for accessing the hardware and a global table, which holds pointers to these functions.

If you are going to use *µC/FS* with you own hardware, you may have to write your own device driver. This chapter describes which functions are required and how to integrate your own device driver into *µC/FS*.



7.1. Device driver functions

In this chapter, you will find a detailed description of the device driver functions required by *µC/FS*. Please note that the names used for these functions are not really relevant for *µC/FS* because the file system accesses those functions through a function table.

7.1.1. _FS_DevIoCtl()

Description

Execute special command on device.

Prototype

```
static int _FS_DevIoCtl(FS_u32 id, FS_i32 cmd, FS_i32 aux,
                        void *buffer);
```

Parameter	Meaning
id	Number of media (0…N)
cmd	Command
aux	Parameter for command
buffer	Pointer to data required by command

Return value

In case of success, the return value is 0. Upon failure, the return value is -1.

Add. information

This function can be used to implement special commands for your device driver. For *µC/FS* the only command that currently needs to be supported is the command FS_CMD_FLUSH_CACHE. This command is used to inform cache logic of the device that all of the cache has to be cleared. If you do not have a cache logic in your device driver, you can simply do nothing and return 0.

Example

```
/* sample taken from the RAM device driver */

static int _FS_DevIoCtl(FS_u32 id, FS_i32 cmd, FS_i32 aux, void *buffer) {
  if (cmd==FS_CMD_FLUSH_CACHE) {
    /* flush caches */
    return 0;
  }
  return -1;
}
```

7.1.2. _FS_DevRead()

Description

Read block from media

Prototype

```
static int _FS_DevRead(FS_u32 id, FS_u32 block,
                       void *buffer);
```

Parameter	Meaning
id	Number of media (0…N)
block	Block number to be read from the media
buffer	Data buffer to which the data is transferred

Return value

In case of success, the return value is `0`. Upon failure, the return value is `-1`.

Additional information

The function should transfer `0x0200` (i.e. 512) bytes, which is the default value for an MS-DOS/MS-Windows compatible FAT file systems. *µC/FS* can support any block size but, if you use the FAT file system layer, you have to use this default block size.

Example

```
/* sample taken from the RAM device driver */

static int _FS_DevRead(FS_u32 id, FS_u32 block, void *buffer) {
  memcpy(buffer,((char*)_FS_rr_devdata[id].data)+block*_FS_rr_devdata[id].bpu,
        _FS_rr_devdata[id].bpu);
  return 0;
}
```

7.1.3. _FS_DevStatus()

Description

Return current status of your device.

Prototype

```
static int _FS_DevStatus(FS_u32 id);
```

Parameter	Meaning
id	Number of media (0...N)

Return value

The function returns `0` if the device can be accessed. If the media has changed (e.g. a card removed or replaced) and the device can be accessed, the return value has to be `FS_LBL_MEDIACHANGED`. Any value `< 0` is interpreted as an error.

Additional information

The main purpose of this function is to detect a media change. All *µC/FS* file operation calls this function to check, if the device can be accessed.

Example

```
/* sample taken from the RAM device driver */

static int _FS_DevStatus(FS_u32 id) {
  static int online[RR_DEVNUM];

  if (!online[id]) {
    online[id] = 1;
    /* init memory (create BPB) */
    return FS_LBL_MEDIACHANGED;
  }
  return 0;
}
```

7.1.4. _FS_DevWrite()

Description

Write block to media

Prototype

```
static int _FS_DevWrite(FS_u32 id, FS_u32 block,
                        void *buffer);
```

Parameter	Meaning
id	Number of media (0…N)
block	Block number to be written on media
buffer	Pointer to data for transfer to the media.

Return value

In case of success the return value is 0. Upon failure, the return value is -1.

Additional information

The function should transfer 0x0200 (i.e. 512) bytes, which is the default value for an MS-DOS/MS-Windows compatible FAT file systems. *µC/FS* can support any block size but, if you use the FAT file system layer, you have to use this default block size.

Example

```
/* sample taken from the RAM device driver */

static int _FS_DevWrite(FS_u32 id, FS_u32 block, void *buffer) {
  memcpy(((char*)_FS_rr_devdata[id].data)+block*_FS_rr_devdata[id].bpu,buffer,
         _FS_rr_devdata[id].bpu);
  return 0;
}
```

7.2. Device driver function table

To use a device driver with *µC/FS*, a global function table is required, which holds pointers to the device driver functions. Each entry in the table contains five values as shown in the example below.

Example

```
/* sample taken from the RAM device driver */

const FS__device_type FS__ramdevice_driver = {
  "RAMDISK device",
  _FS_DevStatus,
  _FS_DevRead,
  _FS_DevWrite,
  _FS_DevIoCtl
};
```

7.3. Integrate a new device driver

If you want to use your own device driver, you have to tell *µC/FS*, which *device name* you would like to use for your device and which *File System Layer* (currently only FAT is supported) you want to use.

You do this by setting appropriate value for `FS_DEVINFO` in your `FS_conf.h`, which is used to initialize *µC/FS*'s global device information table.

The first parameter is a *device name*, which you want to use for *µC/FS's* API calls.

The second parameter is a pointer to a *File System Layer* function table; currently only FAT is supported.

The third parameter is a pointer to *a Device Driver* function table.

The last parameter is reserved for future use and should be zero.

Example

```
/* Default device is 'smc:', because it is first in the list.
   'mydev:' uses FAT and device driver function table FS__mydevice_driver
*/

#define FS_DEVINFO \
        "smc",    &FS__fat_functable, &FS__smcdevice_driver, 0, \
        "mydev",  &FS__fat_functable, &FS__mydevice_driver , 0
```

If you do not specify a value for `FS_DEVINFO`, *µC/FS* will use the following values as default:

```
#ifndef FS_DEVINFO
  #if FS_USE_SMC_DRIVER
```

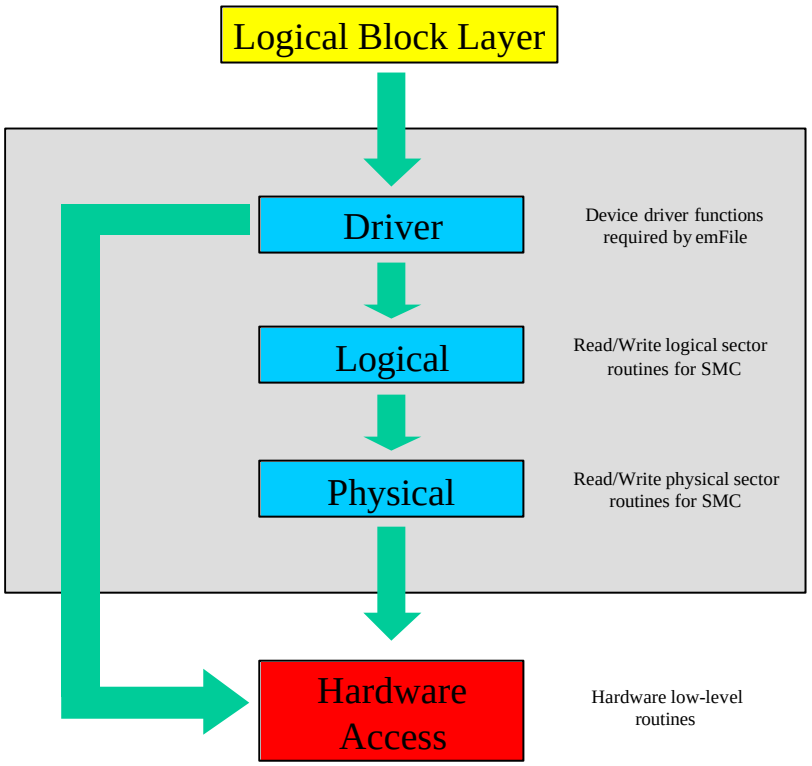
```
    #define FS_DEVINFO_DEVSMC     "smc",    &FS__fat_functable,
&FS__smcdevice_driver, 0,
  #else
    #define FS_DEVINFO_DEVSMC
  #endif
  #if FS_USE_WINDRIVE_DRIVER
    #define FS_DEVINFO_DEVWINDRV  "windrv", &FS__fat_functable,
&FS__windrive_driver,  0,
  #else
    #define FS_DEVINFO_DEVWINDRV
  #endif
  #if FS_USE_RAMDISK_DRIVER
    #define FS_DEVINFO_DEVRAM     "ram",    &FS__fat_functable,
&FS__ramdevice_driver, 0,
  #else
    #define FS_DEVINFO_DEVRAM
  #endif
  #define FS_DEVINFO FS_DEVINFO_DEVSMC FS_DEVINFO_DEVWINDRV FS_DEVINFO_DEVRAM
#endif  /* FS_DEVINFO */
```

The SMC driver will be used as default if it is enabled, since it is first in the table. The last device in the table is the RAM disk driver, which will be the only default device if the SMC and Windows drivers are both disabled.

8. SmartMedia Card Device Driver

µC/FS includes a generic driver for *SmartMedia* cards. To use it in your system, you will have to provide basic I/O functions for accessing your card reader hardware. You can find samples for such routines in the directory “device\smc\hardware”.

The SMC driver is split up into different layers. In the diagram below, there is an overview of these layers. To use the SMC driver with a custom hardware, only a suitable *hardware access* layer is required. All other layers are generic and do not depend on a specific card reader hardware.



This chapter describes all hardware access functions required by *µC/FS*’s generic *SmartMedia* driver.

8.1. Control line functions

8.1.1. FS_SMC_HW_X_BusyLedOff()

Description

Turn off busy LED of the card reader.

Prototype

```
void FS_SMC_HW_X_BusyLedOff(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

Please see FS_SMC_HW_X_BusyLedOn().

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_BusyLedOff(FS_u32 id) {
  if (id==0) {
    __PDDR &= ~0x01;        /* DIAG LED off */
  }
}
```

8.1.2. FS_SMC_HW_X_BusyLedOn()

Description

Turn on busy LED of the card reader.

Prototype

```
void FS_SMC_HW_X_BusyLedOn(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

If your system can lock the card reader, you should also do this here because a call of this function means that a *µC/FS* operation is pending and the card should not be removed until `FS_SMC_HW_X_BusyLedOff()` is called.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_BusyLedOn(FS_u32 id) {
  if (id==0) {
    __PDDR |= 0x01;           /* DIAG LED on */
  }
}
```

8.1.3. FS_SMC_HW_X_SetAddr()

Description

Set *CE* low, *CLE* low and *ALE* high for the specified card reader.

Prototype

```
void FS_SMC_HW_X_SetAddr(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

The generic SMC driver calls this function to start address data transfer.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_SetAddr(FS_u32 id) {
  if (id==0) {
    __PBDR      = 0x6f;       /* nCE low, CLE low, ALE high */
  }
  else if (id==1) {
    __PBDR      = 0xaf;       /* nCE low, CLE low, ALE high  */
  }
}
```

8.1.4. FS_SMC_HW_X_SetCmd()

Description

Set *CE* low, *CLE* high and *ALE* low for the specified card reader.

Prototype

```
void FS_SMC_HW_X_SetCmd(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

The generic SMC driver calls this function to start command transfer.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_SetCmd(FS_u32 id) {
  if (id==0) {
    __PBDR      = 0x5f;       /* nCE low, CLE high, ALE low  */
  }
  else if (id==1) {
    __PBDR      = 0x9f;       /* nCE low, CLE high, ALE low  */
  }
}
```

8.1.5. FS_SMC_HW_X _SetData()

Description

Set *CE* low, *CLE* low and *ALE* low for the specified card reader.

Prototype

```
void FS_SMC_HW_X_SetData(FS_u32 id);
```

Parameter	Meaning
id	Id of card reader (0…N)

Return value

void

Additional information

The generic SMC driver calls this function to start data transfer.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_SetData(FS_u32 id) {
  if (id==0) {
    __PBDR      = 0x4f;       /* nCE low, CLE low, ALE low  */
  }
  else if (id==1) {
    __PBDR      = 0x8f;       /* nCE low, CLE low, ALE low  */
  }
}
```

8.1.6. FS_SMC_HW_X_SetStandby()

Description

Set *CE* high, *CLE* low and *ALE* low for the specified card reader.

Prototype

```
void FS__X_SMC_HW_SetStandby(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

The generic SMC driver calls this function to set the specified card reader into standby mode.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_SetStandby(FS_u32 id) {
  __PBDR      = 0x0f;       /* nCE high, CLE low, ALE low  */
}
```

8.2. Power control functions

8.2.1. FS_SMC_HW_X_VccOff()

Description

Turn off *VCC* for the specified card reader.

Prototype

```
void FS_SMC_HW_X_VccOff(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

If your hardware cannot control *VCC* for the card reader, you can simply leave that function empty. In such case, *VCC* should always be on.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_VccOff(FS_u32 id) {
}
```

8.2.2. FS_SMC_HW_X_VccOn()

Description

Turn on *VCC* for the specified card reader.

Prototype

```
void FS_SMC_HW_X_VccOn(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

If your hardware cannot control *VCC* for the card reader, you can simply leave that function empty. In such case, *VCC* should always be on.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_VccOn(FS_u32 id) {
}
```

8.3. Status detection functions

8.3.1. FS_SMC_HW_X_ChkBusy()

Description

Check if card reader is busy.

Prototype

```
char FS_SMC_HW_X_ChkBusy(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0`, if the card reader is not busy. Any other value does mean, that an operation is pending.

Additional information

If your hardware allows you to monitor the *RY/BY* line, you can use the status of that line. In case your hardware needs to make a device operation to detect if the card reader is busy, you must make sure, that the macro `FS_SMC_HW_SUPPORT_BSYLINE_CHECK` is set to zero in your `fs_conf.h`. This allows you to tell the generic driver not to use `FS_SMC_HW_X_ChkBusy()` when a device operation is pending. Instead, it will wait for the maximum allowed time for that operation.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_ChkBusy(FS_u32 id) {
  char x;

  /* hardware cannot check RY/BY, therefore we use read status */
  FS_SMC_HW_X_SetCmd(id);        /* nCE low, CLE high, ALE low  */
  __SMCPORT   = 0x70;           /* cmd read status   */
  FS_SMC_HW_X_SetData(id);       /* nCE low, CLE low, ALE low  */
  x  = __SMCPORT;
  FS_SMC_HW_X_SetStandby(id);    /* nCE high, CLE low, ALE low  */
  if (x&0x40) {
    return 0;
  }
  return 1;
}
```

8.3.2. FS_SMC_HW_X_ChkCardIn()

Description

The generic SMC driver uses this function to detect, if there is a card in the reader.

Prototype

```
char FS_SMC_HW_X_ChkCardIn(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0`, if there is no card in the reader. Any other value does mean means that there is a card in your reader.

Additional information

If your hardware can directly check the presence of a card, you can use that mechanism and return the proper status. In case your hardware has to try to do a device operation for checking the presence of a card, you must not do that operation inside this function because it is called during other device operations, which would be interrupted. Instead, use a variable to keep track of the current status and, return the value of that variable.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_ChkCardIn(FS_u32 id) {
  return _Hw_card_in[id];
}
```

8.3.3. FS_SMC_HW_X_ChkPower()

Description

Check power is applied to card reader.

Prototype

```
char FS_SMC_HW_X_ChkPower(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0`, if there is a problem with the power supply. Any other value means that the power status is correct.

Additional information

This function is used to detect the result of turning on *VCC*. If your hardware does not support power check, you should simply return a non-zero value.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_ChkPower(FS_u32 id) {
  return 1;
}
```

8.3.4. FS_SMC_HW_X_ChkStatus()

Description

This function detects if the status of the card reader has changed since its last call. It is used by the generic driver to detect a disk change.

Prototype

```
char FS_SMC_HW_X_ChkStatus(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0`, if there is no status change since last call. Any other value does means that status has changed since the last call.

Additional information

For hardware that remembers a disk change, you can simply return that status. If your hardware does not remember a disk change, you will have to check periodically if a card is present in your reader. Whenever you detect a change during that periodic check, you have to make sure, that next call of `FS_SMC_HW_X_ChkStatus()` returns a non-zero value. See also `FS_SMC_HW_X_DetectStatus()` for more information about disk change detection.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_ChkStatus(FS_u32 id) {
  if (_Hw_card_changed[id]) {
    _Hw_card_changed[id] = 0;
    return 1;
  }
  return 0;
}
```

8.3.5. FS_SMC_HW_X_ChkWP()

Description

Check if the current media is write-protected.

Prototype

```
char FS_SMC_HW_X_ChkWP(FS_u32 id);
```

Parameter	Meaning
Id	id of card reader (0…N)

Return value

Returns `0` if the media is not write-protected. Any other value means that the media is write-protected.

Additional information

If your hardware allows to monitor write-protect status of a media directly, you can use this status. In case your hardware has to execute a status read command for checking write protect status of a card, you must not do that operation inside this function because it is called during other device operations that would be interrupted. Instead, use a variable to keep track of the current status instead and return the value of that variable.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_ChkWP(FS_u32 id) {
  return _Hw_card_wprotect[id];
}
```

8.3.6. FS_SMC_HW_X_DetectStatus()

Description

Check if card is present.

Prototype

```
char FS_SMC_HW_X_DetectStatus(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0` if there is a card present and, a non-zero value if there is no card in reader or, the card cannot be accessed.

Additional information

In contrast to `FS_SMC_HW_X_ChkCardIn()`, this function is not called when a device operation is pending and therefore, it is allowed to use a device command for status detection. The generic *SmartMedia* driver calls this function in two cases:

1) Before a µ*C/FS* operation takes place, the file system asks the device driver for current status of the device.

2) During the periodic disk change detect, if your hardware does not remember disk changes. If your hardware cannot check for write-protect status and disk presence without a device operation, you should also make those detections here and remember their status in variables. See also `FS_SMC_HW_X_ChkCardIn()`, `FS_SMC_HW_X_ChkStatus()` and `FS_SMC_HW_X_ChkWP()`.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_DetectStatus(FS_u32 id) {
  char x;

  FS_SMC_HW_X_SetCmd(id);        /* nCE low, CLE high, ALE low  */
  __SMCPORT   = 0x70;             /* cmd read status   */
  FS_SMC_HW_X_SetData(id);       /* nCE low, CLE low, ALE low  */
  x  = __SMCPORT;
  FS_SMC_HW_X_SetStandby(id);    /* nCE high, CLE low, ALE low  */
  if ((x==0x70) || (x&0x01) || (!(x&0x40))) {
    _Hw_card_changed[id]  = 1;
    _Hw_card_wprotect[id] = 1;
    _Hw_card_in[id]       = 0;
    return 1;
  }
  _Hw_card_in[id]         = 1;
  if (x&0x80) {
    _Hw_card_wprotect[id] = 0;
  }
  else {
    _Hw_card_wprotect[id] = 1;
  }
  return 0;
}
```

8.4. Data transfer functions

8.4.1. FS_SMC_HW_X_InData()

Description

For the specified card reader, set *RE* to low, read the lines *I/O1 ~ I/O8* and set *RE* to high.

Prototype

```
char FS_SMC_HW_X_InData(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Data received from the card reader.

Additional information

Usually, you will not have to take care about handling of the *RE* line, because it is done automatically by hardware. If you have to control *RE* line, make sure that timing is according to *SmartMedia* card specification.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

char FS_SMC_HW_X_InData(FS_u32 id) {
  return __SMCPORT;
}
```

8.4.2. FS_SMC_HW_X_OutData()

Description

For the selected card reader, set lines *I/O1 ~ I/O8* to the value specified with *data* and then set *WE* to low. After a write pulse width, set *WE* back to high.

Prototype

```
void FS_SMC_HW_X_OutData(FS_u32 id,unsigned char data);
```

Parameter	Meaning
id	id of card reader (0…N)
data	value to be set on lines *I/O1 ~ I/O8*

Return value

void

Additional information

Usually, you will not have to take care about handling of the *WE* line, because it is done automatically by hardware. If you have to control *WE* line, make sure that timing is according to the *SmartMedia* card specification.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_OutData(FS_u32 id,unsigned char data) {
  __SMCPORT = (char) data;
}
```

8.5. Timer functions

8.5.1. FS_SMC_HW_X_ChkTimer()

Description

Check the current status of the countdown timer for the specified reader *id*.

Prototype

```
int FS_SMC_HW_X_ChkTimer(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

Return `0`, if the time set with `FS_SMC_HW_X_SetTimer()` for the specified reader *id* has expired and, any non-zero value, if it has not expired.

Additional information

See `FS_SMC_HW_X_SetTimer()`.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

int FS_SMC_HW_X_ChkTimer(FS_u32 id) {
  short int x;
  x = __TC2D;
  if (x<0) {
    x=0;
  }
  return x;
}
```

8.5.2. FS_SMC_HW_X_SetTimer()

Description

Set the countdown timer of reader *id* to a time of *time* * 0.1 msec and return.

Prototype

```
void FS_SMC_HW_X_SetTimer(FS_u32 id,unsigned short time);
```

Parameter	Meaning
id	id of card reader (0…N)
time	time in 0.1msec units

Return value

void

Additional information

Unlike `FS_SMC_HW_X_WaitTimer()`, this function does not wait until the time has elapsed before returning. Instead, it returns immediately and the driver can check with `FS_SMC_HW_X_ChkTimer()` to see if the time has elapsed. See also `FS_SMC_HW_X_WaitTimer()` for additional information.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_Set_Timer(FS_u32 id,unsigned short time) {
  __INTMR1  &= ~0x200ul;     /* disable TC2 underflow interrupt */
  __SYSCON1 |= 0x80ul;       /* Set clock in 512kHz mode */
  __SYSCON1 &= ~ 0x40ul;     /* free running */
  __TC2D     = time*50;      /* Load counter */
}
```

8.5.3. FS_SMC_HW_X_StopTimer()

Description

Stop the countdown timer of the specified reader *id*.

Prototype

```
void FS_SMC_HW_X_StopTimer(FS_u32 id);
```

Parameter	Meaning
id	id of card reader (0…N)

Return value

void

Additional information

See `FS_SMC_HW_X_SetTimer()`.

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_StopTimer(FS_u32 id)  {
}
```

8.5.4. FS_SMC_HW_X_WaitTimer()

Description

Wait for a time of *time* * 0.1 msec and return.

Prototype

```
void FS_SMC_HW_X_WaitTimer(FS_u32 id,unsigned short time);
```

Parameter	Meaning
id	id of card reader (0…N)
time	time in 0.1msec units

Return value

void

Additional information

The generic *SmartMedia* card driver uses this function to wait for a device operation to complete. If you have more than one card reader in your system and, you can access them simultaneously, you will need a separate timer for each reader. If your card readers share resources (e.g. the same I/O pins), one timer is sufficient because you will have to avoid simultaneous card reader operations .

Example

```
/* sample taken from 'device\smc\Hardware\EP7312' */

void FS_SMC_HW_X_WaitTimer(FS_u32 id,unsigned short time) {
  short x;
  __INTMR1  &= ~0x200ul;     /* disable TC2 underflow interrupt */
  __SYSCON1 |= 0x80ul;       /* Set clock in 512kHz mode */
  __SYSCON1 &= ~ 0x40ul;     /* free running */
  __TC2D     = time*50;      /* Load counter */
  do {
    x = __TC2D;
  } while (x>0);
}
```

9. *MultiMedia & SD* card device driver

µC/FS includes a generic driver for *MultiMedia & SD* cards. The driver accesses cards using SPI mode. For details on *MultiMedia & SD* cards, please check their specifications, which are available at the following websites:

http://www.mmca.org/
http://www.sdcard.org/

To use the driver in your system, you will have to provide basic I/O functions for accessing your card reader hardware. You can find samples of these routines in the directory `device\mmc_sd\hardware`.

This chapter describes all hardware access functions required by *µC/FS*'s generic *MultitMedia & SD* card driver. The sample code, which can be found in the description, uses normal port pins instead of a real SPI. Therefore, data transfer functions look a bit complex.

9.1. Control line functions

9.1.1. FS_MMC_HW_X_BusyLedOff()

Description

Turns off busy LED of the card reader.

Prototype

```
void FS_MMC_HW_X_BusyLedOff(FS_u32 id);
```

Parameter	Meaning
`id`	ID of card reader (0…N).

Return value

Void.

Additional information

Please see `FS_MMC_HW_X_BusyLedOn()`.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

void FS_MMC_HW_X_BusyLedOff(FS_u32 id) {
  /* no LED available */
}
```

9.1.2. FS_MMC_HW_X_BusyLedOn()

Description

Turns on busy LED of the card reader.

Prototype

```
void FS_MMC_HW_X_BusyLedOn(FS_u32 id);
```

Parameter	Meaning
Id	ID of card reader (0…N).

Return value

Void.

Additional information

If your system can lock the card reader, you should also see that this is done here, because a call of this function means that an *µC/FS* operation is pending and the card should not be removed until `FS_MMC_HW_X_BusyLedOff()` is called.

Example

```
/* sample taken from 'device\smc\Hardware\ M16C_137x_IP' */

void FS_MMC_HW_X_BusyLedOn(FS_u32 id) {
  /* no LED available */
}
```

9.1.3. FS_MMC_HW_X_ClockCard()

Description

Clocks the card 8*`num` times while data line is high.

Prototype

```
void FS_MMC_HW_X_ClockCard(FS_u32 id, int num);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`num`	Number of clocks/8.

Return value

Void.

Additional information

If you have a real SPI in your system, you can simply send a 0xff for `num` times.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

void FS_MMC_HW_X_ClockCard(FS_u32 id, int num) {
  if (id!=0) {
    return;
  }
  num *=8;
  SPI_SET_DATAOUT;
  while (num) {
    SPI_CLR_CLK;
    SPI_DELAY;
    SPI_SET_CLK;
    SPI_DELAY;
    num--;
  }
}
```

9.1.4. FS_MMC_HW_X_SetCS()

Description

Sets CS signal of the specified card reader.

Prototype

```
void FS_MMC_HW_X_SetCS(FS_u32 id, char high);
```

Parameter	Meaning
id	ID of card reader (0…N).
high	1 means high, 0 means low.

Return value

Void.

Additional information

CS signal is used to address a specific card reader connected to the SPI.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

void FS_MMC_HW_X_SetCS(FS_u32 id, char high) {
  if (id!=0) {
    return;
  }
  if (high) {
    SPI_SET_CS;
  }
  else {
    SPI_CLR_CS;
  }
}
```

9.2. Operation condition detection and adjusting functions

9.2.1. FS_MMC_HW_X_AdjustFOP()

Description

Adjusts host to card clock frequency according to `tran_speed`.

Prototype

```
char FS_MMC_HW_X_AdjustFOP(FS_u32 id, FS_u32 tran_speed);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`tran_speed`	Lower 8 bits contain TRAN_SPEED parameter of the CSD.

Return value

0 if no problem occurs. If the host cannot adjust a communication speed lower than or equal to what is specified with the TRAN_SPEED parameter, a non-zero value will be returned.
Please refer the *MultiMedia & SD* card specifications on how to interpret the TRANS_SPEED parameter.

Additional information

µC/FS's generic *MultiMedia & SD* card driver calls this function after reading TRAN_SPEED of the card's CSD register, allowing the host to adjust the communication speed according to the card's limitations. Before this function is called, you should not access a card faster than 400KHz.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

char FS_MMC_HW_X_AdjustFOP(FS_u32 id, FS_u32 tran_speed) {
  return 0;
}
```

9.2.2. FS_MMC_HW_X_CheckOCR()

Description

Checks and adjusts operation voltage.

Prototype

```
char FS_MMC_HW_X_CheckOCR(FS_u32 id, FS_u32 ocr);
```

Parameter	Meaning
id	ID of card reader (0…N).
ocr	Value of the OCR register.

Return value

0 if the host can operate the card at requested voltage.
Any other value means that the host cannot work at the requested voltage.

Additional information

Please check the *MultiMedia & SD* card specifications for details.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

char FS_MMC_HW_X_CheckOCR(FS_u32 id, FS_u32 ocr) {
  if (id!=0) {
    return 1;
  }
  if (ocr&0x003c0000) {
    /* we support 3.0 - 3.4 Volt */
    return 0;
  }
  return 1;
}
```

9.2.3. FS_MMC_HW_X_GetFOP()

Description

Returns current host to card clock frequency.

Prototype

```
FS_u32 FS_MMC_HW_X_GetFOP(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Current host to card clock frequency.

Additional information

This value is used to calculate the absolute time dependent part of card access timing.
Please also check the *MultiMedia & SD* card specifications for details.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

FS_u32 FS_MMC_HW_X_GetFOP(FS_u32 id) {
  return FS_MMC_CLOCK_FREQ;
}
```

9.3. Status detection functions

9.3.1. FS_MMC_HW_X_CheckWP()

Description

Checks the status of the mechanical write protect switch of an *SD* card.

Prototype

```
char FS_MMC_HW_X_CheckWP(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

0 if the card is not write protected.
Any other value means that the card is write protected.

Additional information

MultiMedia cards do not have a mechanical write protect switch and should return 0. If you are using an *SD* card, please be aware that the mechanical switch does not really protect the card; it is the responsibility of the host to respect the status of that switch.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

char FS_MMC_HW_X_CheckWP(FS_u32 id) {
  return 0;
}
```

9.3.2. FS_MMC_HW_X_DetectStatus()

Description

Checks whether a card is present.

Prototype

```
char FS_MMC_HW_X_DetectStatus(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

0 if there is a card present.
Any other value means there is no card in the reader or the card cannot be accessed.

Additional information

Usually, a card reader provides a hardware signal which can be used. The sample code below is for a specific hardware which does not have such a signal. Therefore, the presence of a card is checked by executing a card command.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

char FS_MMC_HW_X_DetectStatus(FS_u32 id) {
  static int init;

  if (id!=0) {
    return 1;
  }
  if (!init) {
      PD6  = 0xb0;      //  Direction register   (0=IN 1=OUT)
      P6   = 0xb0;
      init = 1;
  }
  FS_MMC_HW_X_SetCS(id,0);
  if (FS__MMC_CheckCardCSD(id)!=0) {
    /* Reading card parameters failed; try complete RESET sequence. */
    if (FS__MMC_Init(id)!=0) {
      FS_MMC_HW_X_SetCS(id,1);
      return 1;
    }
  }
  FS_MMC_HW_X_SetCS(id,1);
  return 0;
}
```

9.3.3. FS_MMC_HW_X_WaitBusy()

Description

Waits for a maximum of 8*`maxwait` clocks for a card to become ready.

Prototype

```
unsigned char FS_MMC_HW_X_WaitBusy(FS_u32 id, FS_u32
maxwait);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`maxwait`	Maximum number of clocks/8 before timeout.

Return value

0 if the card is no longer busy.
A non-zero value means the card is still busy.

Additional information

When using a real SPI, you can simply read from it until you receive a non-zero value or a timeout occurred.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

unsigned char  FS_MMC_HW_X_WaitBusy(FS_u32 id, FS_u32 maxwait) {
  unsigned char a;

  if (id!=0) {
    return 0xff;
  }
  SPI_SET_DATAOUT;
  maxwait *= 8;
  /* Wait until timeout or data line is high */
  do {
    SPI_CLR_CLK;
    SPI_DELAY;
    a = SPI_DATAIN;
    SPI_SET_CLK;
    SPI_DELAY;
    if (a) {
      break;
    }
    if (maxwait) {
      maxwait--;
    }
  } while (maxwait);
  if (!a) {
    /* still busy */
    return 0xff;
  }
  return 0;
}
```

9.4. Data transfer functions

9.4.1. FS_MMC_HW_X_ReadByte()

Description

Reads the next byte starting with a 0 bit from the SPI.

Prototype

```
unsigned char FS_MMC_HW_X_ReadByte(FS_u32 id, FS_u32
maxwait);
```

Parameter	Meaning
id	ID of card reader (0…N).
maxwait	Maximum number of clocks/8 before timeout.

Return value

The next byte with bit 7 = 0, or 0xff if timeout occurred.

Additional information

This function is used to receive card responses which have the highest bit set to 0.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

unsigned char FS_MMC_HW_X_ReadByte(FS_u32 id, FS_u32 maxwait) {
  unsigned char a,bpos;
  if (id!=0) {
    return 0xff;
  }
  SPI_SET_DATAOUT;
  maxwait *= 8;
  /* sync on 1st 0 bit */
  do {
    SPI_CLR_CLK;
    SPI_DELAY;
    a = SPI_DATAIN;
    SPI_SET_CLK;
    SPI_DELAY;
    if (!a) {
      break;
    }
    if (maxwait) {
      maxwait--;
    }
  } while (maxwait);
  if (a) {
    return 0xff;
  }
  /* get 7 remaining bits */
  bpos = 7;
  do {
    bpos--;
    SPI_CLR_CLK;
    SPI_DELAY;
    a |= (SPI_DATAIN << bpos);
    SPI_SET_CLK;
    SPI_DELAY;
  } while (bpos);
  return a;
}
```

9.4.2. FS_MMC_HW_X_ReadByteNoSync()

Description

Reads the next byte from the SPI.

Prototype

```
unsigned char FS_MMC_HW_X_ReadByteNoSync(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

The next byte received from the SPI.

Additional information

This function is used to read normal data from the card. If your system has a usual SPI, you can simply read from it.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

unsigned char FS_MMC_HW_X_ReadByteNoSync(FS_u32 id) {
  unsigned char a,bpos;

  if (id!=0) {
    return 0xff;
  }
  SPI_SET_DATAOUT;
  a = 0;
  /* get 8 bits */
  bpos = 8;
  do {
    bpos--;
    SPI_CLR_CLK;
    SPI_DELAY;
    a |= (SPI_DATAIN << bpos);
    SPI_SET_CLK;
    SPI_DELAY;
  } while (bpos);
  return a;
}
```

9.4.3. FS_MMC_HW_X_ReadSingleBlock()

Description

Reads a data block from the SPI.

Prototype

```
unsigned char FS_MMC_HW_X_ReadSingleBlock(FS_u32 id,
unsigned char *buf, int len, FS_u32 maxwait);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`buf`	Pointer to a buffer for data to receive.
`len`	Number of bytes to receive.
`maxwait`	Maximum number of clocks/8 before timeout.

Return value

0 on success; 0xff in case of a problem.

Additional information

This function is used to read a complete data block starting with a *data token* and ending with a *CRC*. Only the data in between is stored to the data buffer. By default, the *CRC* is not checked in SPI mode; please also see the *MultiMedia & SD* card specifications for details.

When using a real SPI, you can simply read from it until you get a valid *data token* or timeout occurs. Then read the real data and the *CRC*. Make sure that you do not store the *data token* or the *CRC* to the buffer.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

unsigned char FS_MMC_HW_X_ReadSingleBlock(FS_u32 id, unsigned char *buf, int
len, FS_u32 maxwait) {
  unsigned char a,bpos;
  int i;
  if (id!=0) {
    return 0xff;
  }
  SPI_SET_DATAOUT;
  maxwait *= 8;
  /* sync on 1st 0 bit (LSB in data token 'Start Block') */
  do {
    SPI_CLR_CLK;
    SPI_DELAY;
    a = SPI_DATAIN;
    SPI_SET_CLK;
    SPI_DELAY;
    if (!a) {
      break;
    }
    if (maxwait) {
      maxwait--;
    }
  } while (maxwait);
  if (a) {
    return 0xff;
  }
  /* read data + 16 bit CRC */
  for (i=0;i<(len+2);i++) {
    bpos = 8;
    a = 0;
```

```
    do {
      bpos--;
      SPI_CLR_CLK;
      SPI_DELAY;
      a |= (SPI_DATAIN << bpos);
      SPI_SET_CLK;
      SPI_DELAY;
    } while (bpos);
    if (i<len) {
      buf[i]=a;
    }
  }
  return 0;
}
```

9.4.4. FS_MMC_HW_X_WriteByte()

Description

Sends data to the card.

Prototype

```
void FS_MMC_HW_X_WriteByte(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Data to send.

Return value

Void.

Additional information

When using a real SPI, you can simply send the data.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */

void FS_MMC_HW_X_WriteByte(FS_u32 id, unsigned char data) {
  if (id!=0) {
    return;
  }
  _FS_MMC_HW_Send1(data);
}
```

9.4.5. FS_MMC_HW_X_WriteByteSingleBlock()

Description

Sends a data block to the card.

Prototype

```
void FS_MMC_HW_X_WriteSingleBlock(FS_u32 id, unsigned char
*buf, int len);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`buf`	Pointer to data to send.
`len`	Number of bytes to send.

Return value

Void.

Additional information

This function is used to send a complete data block to the card. A data block starts with a *data token* and ends with a *CRC*.
By default, the *CRC* is ignored in SPI mode and any value can be sent instead.

Example

```
/* sample taken from 'device\smc\Hardware\M16C_137x_IP' */


void FS_MMC_HW_X_WriteSingleBlock(FS_u32 id, unsigned char *buf, int len) {
  int i;

  if (id!=0) {
    return;
  }
  SPI_SET_DATAOUT;
  _FS_MMC_HW_Send1(0xfe);       /* Start Block (Single Block Write) */
  for (i=0;i<len;i++) {
    _FS_MMC_HW_Send1(buf[i]);
  }
  /* Send dummy CRC (by default not checked in SPI mode) */
  _FS_MMC_HW_Send1(0xff);
  _FS_MMC_HW_Send1(0xff);
  SPI_SET_DATAOUT;
}
```

10. *CompactFlash* card & IDE device driver

µC/FS's generic *CompactFlash* & IDE device driver can be used to access usual ATA HD drives or *CompactFlash* storage cards also known as CF using *true IDE* mode. For details on *CompactFlash*, please check the specification, which is available at:

http://www.compactflash.org/

Information about the *AT Attachment* interface can be found at the *Technical Committee T13*, who is responsible for the ATA standard:

http://www.t13.org/

To use the driver with your specific hardware, you will have to provide basic I/O functions for accessing the ATA I/O registers. This chapter describes all these routines.

10.1. Control line functions

10.1.1. FS_IDE_HW_X_BusyLedOff()

Description

Turns off busy LED of the card reader.

Prototype

```
void FS_IDE_HW_X_BusyLedOff(FS_u32 id);
```

Parameter	Meaning
`id`	ID of card reader (0…N).

Return value

Void.

Additional information

Please see `FS_IDE_HW_X_BusyLedOn()`.

Example

```
/* sample taken from 'device\ide\Hardware\EP7312' */

void FS_IDE_HW_X_BusyLedOff(FS_u32 id) {
  /* no LED available */
}
```

10.1.2. FS_IDE_HW_X_BusyLedOn()

Description

Turns on busy LED of the card reader.

Prototype

```
void FS_IDE_HW_X_BusyLedOn(FS_u32 id);
```

Parameter	Meaning
`id`	ID of card reader (0…N).

Return value

Void.

Additional information

If your system can lock the card reader or device, you should also see that this is done here, because a call of this function means that an *µC/FS* operation is pending and the card or device should not be removed until `FS_IDE_HW_X_BusyLedOff()` is called.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_BusyLedOn(FS_u32 id) {
  /* no LED available */
}
```

10.1.3. FS_IDE_HW_X_HWReset()

Description

Reset the bus interface.

Prototype

```
void FS_IDE_HW_X_HWReset(FS_u32 id);
```

Parameter	Meaning
`id`	ID of card reader (0…N).

Return value

Void.

Additional information

This function is called, if `FS_IDE_HW_X_DetectStatus()` detects, that a new device or card is present. For usual ATA devices, the function can be empty.

If you are going to use a *CompactFlash* card, make sure that the bus is power cycled in this function while ~OE is grounded, so that the card operates in *true IDE* mode. This is **important**, because a CF card inserted in a normal card reader will be provided with VCC and GND before ~OE is connected. That will cause the card to run in *PC Card ATA* mode, which is not supported by the driver.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_HWReset(FS_u32 id) {
}
```

10.2. ATA I/O register access functions

10.2.1. FS_IDE_HW_X_GetAltStatus()

Description

Read the *Alternate Status* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetAltStatus(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Alternate Status* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetAltStatus(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_DC;
  return data;
}
```

10.2.2. FS_IDE_HW_X_GetCylHigh()

Description

Read the *Cylinder High* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetCylHigh(FS_u32 id);
```

Parameter	Meaning
`id`	ID of card reader (0…N).

Return value

Value of the *Cylinder High* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetCylHigh(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_CH;
  return data;
}
```

10.2.3. FS_IDE_HW_X_GetCylLow()

Description

Read the *Cylinder Low* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetCylLow(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Cylinder Low* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetCylLow(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_CL;
  return data;
}
```

10.2.4. FS_IDE_HW_X_GetData()

Description

Read the *RD Data* register.

Prototype

```
FS_u16 FS_IDE_HW_X_GetData(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *RD Data* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

FS_u16 FS_IDE_HW_X_GetData(FS_u32 id) {
  FS_u16 data;

  __MEMCFG2   = 0x1c13;     /* CS5 16 bit */
  data = __IDE_DATA;
  return data;
}
```

10.2.5. FS_IDE_HW_X_GetDevice()

Description

Read the *Device/Head* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetDevice(FS_u32 id);
```

Parameter	Meaning
Id	ID of card reader (0…N).

Return value

Value of the *Device/Head* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetDevice(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_DH;
  return data;
}
```

10.2.6. FS_IDE_HW_X_GetError()

Description

Read the *Error* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetError(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Error* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetError(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_FC;
  return data;
}
```

10.2.7. FS_IDE_HW_X_GetSectorCount()

Description

Read the *Sector Count* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetSectorCount(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Sector Count* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetSectorCount(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_SC;
  return data;
}
```

10.2.8. FS_IDE_HW_X_GetSectorNo()

Description

Read the *Sector Number* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetSectorNo(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Sector Number* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetSectorNo(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_SN;
  return data;
}
```

10.2.9. FS_IDE_HW_X_GetStatus()

Description

Read the *Status* register.

Prototype

```
unsigned char FS_IDE_HW_X_GetStatus(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

Value of the *Status* register.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

unsigned char FS_IDE_HW_X_GetStatus(FS_u32 id) {
  unsigned char data;

  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  data = __IDE_CMD;
  return data;
}
```

10.2.10. FS_IDE_HW_X_SetCommand()

Description

Set the *Command* register.

Prototype

```
void FS_IDE_HW_X_SetCommand(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Command* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetCommand(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_CMD    = data;
}
```

10.2.11. FS_IDE_HW_X_SetCylHigh()

Description

Set the *Cylinder High* register.

Prototype

```
void FS_IDE_HW_X_SetCylHigh(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Cylinder High* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetCylHigh(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_CH    = data;
}
```

10.2.12. FS_IDE_HW_X_SetCylLow()

Description

Set the *Cylinder Low* register.

Prototype

```
void FS_IDE_HW_X_SetCylLow(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Cylinder Low* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetCylLow(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_CL    = data;
}
```

10.2.13. FS_IDE_HW_X_SetData()

Description

Set the *WR Data* register.

Prototype

```
void FS_IDE_HW_X_SetData(FS_u32 id, FS_u16 data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *WR Data* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetData(FS_u32 id, FS_u16 data) {
  __MEMCFG2   = 0x1c13;     /* CS5 16 bit */
  __IDE_DATA  = data;
}
```

10.2.14. FS_IDE_HW_X_SetDevControl()

Description

Set the *Device Control* register.

Prototype

```
void FS_IDE_HW_X_SetDevControl(FS_u32 id, unsigned char
data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Device Control* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetDevControl(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_DC    = data;
}
```

10.2.15. FS_IDE_HW_X_SetDevice()

Description

Set the *Device/Head* register.

Prototype

```
void FS_IDE_HW_X_SetDevice(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Device/Head* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetDevice(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_DH    = data;
}
```

10.2.16. FS_IDE_HW_X_SetFeatures()

Description

Set the *Features* register.

Prototype

```
void FS_IDE_HW_X_SetFeatures(FS_u32 id, unsigned char
data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Features* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetFeatures(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_FC    = data;
}
```

10.2.17. FS_IDE_HW_X_SetSectorCount()

Description

Set the *Sector Count* register.

Prototype

```
void FS_IDE_HW_X_SetSectorCount(FS_u32 id, unsigned char data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Sector Count* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetSectorCount(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_SC    = data;
}
```

10.2.18. FS_IDE_HW_X_SetSectorNo()

Description

Set the *Sector Number* register.

Prototype

```
void FS_IDE_HW_X_SetSectorNo(FS_u32 id, unsigned char
data);
```

Parameter	Meaning
`id`	ID of card reader (0…N).
`data`	Value to write to the *Sector Number* register

Return value

Void.

Additional information

Please check *AT Attachment* interface specification or *CompactFlash* specification for details.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

void FS_IDE_HW_X_SetSectorNo(FS_u32 id, unsigned char data) {
  __MEMCFG2   = 0x1f13;     /* CS5 8 bit */
  __IDE_SN    = data;
}
```

10.3. Status detection functions

10.3.1. FS_IDE_HW_X_DetectStatus()

Description

Checks whether a CF card or ATA device is present.

Prototype

```
char FS_IDE_HW_X_DetectStatus(FS_u32 id);
```

Parameter	Meaning
id	ID of card reader (0…N).

Return value

0 if there is a card or device present and can be accessed.
Any other value means there is no card in the reader or the device cannot be accessed.

Additional information

The sample below is used for accessing a normal ATA device, such as e.g. an IDE hard disk drive. It checks, if you can access the *Sector Count* and *Sector Number* register.

When using the driver for accessing *CompactFlash* cards, you should check lines CD1 and CD2 for presence of a card instead. There are two reasons for that:

1) When a CF card is inserted while VCC is on, the card will be in *PC Card ATA* mode. Therefore you cannot access *Sector Count* and *Sector Number* registers using *true IDE* mode.

2) Checking the lines is usually faster than selecting the device and accessing the *Sector Count* and *Sector Number* registers.

For further details, please check *AT Attachment* interface specification or *CompactFlash* specification.

Example

```
/* sample taken from 'device\ide\Hardware\ep7312' */

char FS_IDE_HW_X_DetectStatus(FS_u32 id) {
  static char init;
  unsigned char a,b;

  if (!init) {
    init  = 1;
    __MEMCFG2  = 0x1f13;        /* CS5 8 bit */
    __SYSCON1 |= 0x40000ul;     /* enable expansion clock */
  }
  if (id==0) {
    FS_IDE_HW_X_SetDevice(id,0xa0);
  }
  else {
    FS_IDE_HW_X_SetDevice(id,0xe0);
  }
  HW__DELAY400NS;
  FS_IDE_HW_X_SetSectorCount(id,0x55);
  FS_IDE_HW_X_SetSectorNo(id,0xaa);
```

```
  FS_IDE_HW_X_SetSectorCount(id,0xaa);
  FS_IDE_HW_X_SetSectorNo(id,0x55);
  FS_IDE_HW_X_SetSectorCount(id,0x55);
  FS_IDE_HW_X_SetSectorNo(id,0xaa);
  a = FS_IDE_HW_X_GetSectorCount(id);
  b = FS_IDE_HW_X_GetSectorNo(id);

  if ((a==0x55) && (b==0xaa)) {
    _HW_DevicePresent[id] = 1;
  }
  else {
    _HW_DevicePresent[id] = 0;
  }
  return (!_HW_DevicePresent[id]);
}
```

11. OS integration

µC/FS is suitable for usage with just about any multithreaded environment. To ensure that different tasks can access the file system concurrently, you have to implement a few operating system dependent functions.

For *embOS, uC/OS-II* and MS-Windows, you will find implementations of these functions in the file system's source code. In this chapter, you will find a description of all the functions that are required to fully support *µC/FS* in multithreaded environments. If you do not use an OS, or if you do not make file access from different tasks, you can implement these functions as empty routines.

You will also add date and time support functions for use by the FAT file system. The example implementations provided with *µC/FS* use ANSI C standard functions to get the correct date and time.

11.1. OS layer control functions

11.1.1. FS_X_OS_Exit()

Description

Delete OS resources. Specifically, you need to delete the binary semaphores used by the file system.

Prototype

```
int FS_X_OS_Exit(void);
```

Return value

In case of success return value is `0`. On failure return value is `-1`.

Additional information

This function is called by `FS_Exit()`. You should delete all resources required by the OS to support multithreading of the file system.

Example

```
/* sample for µC/OS-II */

int  FS_X_OS_Exit (void)
{
    INT8U  err;


    OSSemDel(FS_SemFileHandle, OS_DEL_ALWAYS, &err);
    OSSemDel(FS_SemFileOps   , OS_DEL_ALWAYS, &err);
    OSSemDel(FS_SemMemManager, OS_DEL_ALWAYS, &err);
    OSSemDel(FS_SemDeviceOps , OS_DEL_ALWAYS, &err);
    return (0);
}
```

11.1.2. FS_X_OS_Init()

Description

Initializes the OS resources. Specifically, you will need to create four binary semaphores.

Prototype

```
int FS_X_OS_Init(void);
```

Return value

In case of success return value is `0`. On failure return value is `-1`.

Additional information

This function is called by `FS_Init()`. You should create all resources required by the OS to support multithreading of the file system.

Example

```
/* sample for µC/OS-II  */

int  FS_X_OS_Init (void)
{
    FS_SemFileHandle = OSSemCreate(1);
    FS_SemFileOps    = OSSemCreate(1);
    FS_SemMemManager = OSSemCreate(1);
    FS_SemDeviceOps  = OSSemCreate(1);
    return (0);
}
```

11.2. Internal data structure protection

11.2.1. FS_X_OS_LockFileHandle()

Description

Lock file handle table.

Prototype

```
void FS_X_OS_LockFileHandle(void);
```

Return value

void

Additional information

µC/FS has a table for all open files. The OS integration has to make sure that only one task can access this table at the same time. This is done by waiting on a binary semaphore.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_LockFileHandle (void)
{
    INT8U  err;


    OSSemPend(FS_SemFileHandle, 0, &err);
}
```

11.2.2. FS_X_OS_LockMem()

Description

Lock FAT memory block table.

Prototype

```
void FS_X_OS_LockMem(void);
```

Return value

void

Additional information

The FAT file system layer of *µC/FS* has an internal memory manager. This function is to make sure that only one thread allocates a memory block at any given time.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_LockMem (void)
{
    INT8U  err;

    OSSemPend(FS_SemMemManager, 0, &err);
}
```

11.2.3. FS_X_OS_UnlockFileHandle()

Description

Unlock file handle table.

Prototype

```
void FS_X_OS_UnlockFileHandle(void);
```

Return value

void

Additional information

µC/FS has a table for all open files. The OS integration has to make sure that only one task can access this table at any given time. This function releases the binary semaphore used for that purpose.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_UnlockFileHandle (void)
{
    OSSemPost(FS_SemFileHandle);
}
```

11.2.4. FS_X_OS_UnlockMem()

Description

Unlock FAT memory block table.

Prototype

```
void FS_X_OS_UnlockMem(void);
```

Return value

void

Additional information

The FAT file system layer of *µC/FS* has a memory manager. This function releases access to the memory manager.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_UnlockMem (void)
{
    OSSemPost(FS_SemMemManager);
}
```

11.3. File access protection

11.3.1. FS_X_OS_LockFileOp()

Description

Lock file operation on a specific file.

Prototype

```
void FS_X_OS_LockFileOp(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.

Return value

void

Additional information

Different tasks could access the same file. Therefore an OS implementation, which wants to support concurrent file operations has to lock a file operation. However, if you only allow one file operation at any given time in your application then, you wouldn't need this semaphore. In a multitasking environment, it is however, recommended to use the semaphore mechanism.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_LockFileOp (FS_FILE *fp)
{
    INT8U  err;


    OSSemPend(FS_SemFileOps, 0, &err);
}
```

11.3.2. FS_X_OS_UnlockFileOp()

Description

Unlock file operation on a specific file.

Prototype

```
void FS_X_OS_UnlockFileOp(FS_FILE *fp);
```

Parameter	Meaning
fp	Pointer to a data structure of type FS_FILE.

Return value

void

Additional information

Different tasks could access the same file. Therefore an OS implementation, which wants to support concurrent file operations, has to lock a file operation. However, if you only allow one file operation at any given time in your application then, you wouldn't need this semaphore. In a multitasking environment, it is however, recommended to use the semaphore mechanism.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_UnlockFileOp (FS_FILE *fp)
{
    OSSemPost(FS_SemFileOps);
}
```

11.4. Device access protection

11.4.1. FS_X_OS_LockDeviceOp()

Description

Lock device operation for a specific driver and media.

Prototype

```
void FS_X_OS_LockDeviceOp(const _FS_device_type *driver,
                          FS_u32 id);
```

Parameter	Meaning
`driver`	Pointer to device driver.
`id`	Number of media (0…N).

Return value

void

Additional information

µC/FS allows you to make concurrent device operations. To ensure that only one tasks makes an access to a specific media via a device driver at any given time, this function has to lock an operation by looking to the `driver` and `id`. This is done via another binary semaphore. If your application never accesses multiple devices concurrently then, you don't need this locking mechanism. However, we recommend that you implement this functionality.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_LockDeviceOp (const FS__device_type *driver, FS_u32 id)
{
    INT8U  err;


    OSSemPend(FS_SemDeviceOps, 0, &err);
}
```

11.4.2. FS_X_OS_UnlockDeviceOp()

Description

Unlock device operation for a specific driver and media.

Prototype

```
void FS_X_OS_UnlockDeviceOp(const _FS_device_type *driver,
                            FS_u32 id);
```

Parameter	Meaning
`driver`	Pointer to device driver.
`id`	Number of media (0…N).

Return value

void

Additional information

µC/FS allows you to make concurrent device operations. To ensure that only one tasks makes an access to a specific media via a device driver at any given time, this function has to lock an operation by looking to `driver` and `id`. This is done via another binary semaphore. If your application never accesses multiple devices concurrently then, you don't need this locking mechanism. However, we recommend that you implement this functionality.

Example

```
/* sample for µC/OS-II  */

void  FS_X_OS_UnlockDeviceOp (const FS__device_type *driver, FS_u32 id)
{
    OSSemPost(FS_SemDeviceOps);
}
```

11.5. Time/Date functions

11.5.1. FS_X_OS_GetDate()

Description

Get current date.

Prototype

```
FS_u16  FS_X_OS_GetDate (void);
```

Return value

Return current date as FS_u16 in a format suitable for the FAT file system.

Additional information

The correct date format is:

- Bit 0-4: Day of month (1-31)
- Bit 5-8: Month of year (1-12)
- Bit 9-15: Count of years from 1980 (0-127)

Example

```
/* sample using ANSI C time functions */

FS_u16  FS_X_OS_GetDate (void)
{
#if FS_OS_TIME_SUPPORT == 1
    FS_u16      fdate;
    time_t      t;
    struct tm  *ltime;


    time(&t);
    ltime  = localtime(&t);
    fdate  = ltime->tm_mday;
    fdate += ((FS_u16)(ltime->tm_mon  +  1) << 5);
    fdate += ((FS_u16)(ltime->tm_year - 80) << 9);
#else
    FS_u16      fdate;


    fdate  = 1;
    fdate += ((FS_u16) 1 << 5);
    fdate += ((FS_u16) 0 << 9);
#endif
    return (fdate);
}
```

11.5.2. FS_X_OS_GetTime()

Description

Get current time.

Prototype

```
FS_u16  FS_X_OS_GetTime (void);
```

Return value

Return current time as FS_u16 in a format suitable for the FAT file system.

Additional information

The correct date format is:

- Bit 0-4: 2-second count (0-29)
- Bit 5-10: Minutes (0-59)
- Bit 11-15: Hours (0-23)

Example

```
/* sample using ANSI C time functions */

FS_u16  FS_X_OS_GetTime (void)
{
#if FS_OS_TIME_SUPPORT == 1
    FS_u16      ftime;
    time_t      t;
    struct tm  *ltime;


    time(&t);
    ltime  = localtime(&t);
    ftime  = ltime->tm_sec / 2;
    ftime += ((FS_u16) ltime->tm_min  <<  5);
    ftime += ((FS_u16) ltime->tm_hour << 11);
#else
    FS_u16 ftime;


    ftime  = 0;
#endif
    return (ftime);
}
```

Index

µ

µC/FS
- add directories 15
- add files 15
- API functions for 24–36
- configuration of 15, 17–23
- device drivers for 43–49
- features of 7
- in multithreaded environments 111–22
- installing 10
- integrating into your system 15–16
- layers of 8
- OS integration with 111–22
- sample project 10–14
- using with SMC device driver 50–71

A

ANSI 6, 7
API functions 24–36
- direct input/output 30–31
- error-handling 34–35
- file access 27–29
- file positioning 32–33
- file system control 25–26
- operations on files 36

API layer, of µC/FS 8

B

Building your application 15

C

C programming language 6
Configuration, of µC/FS 17–23

D

Device driver functions
- _FS_DevIoCtl 44
- _FS_DevRead 45
- _FS_DevStatus 46
- _FS_DevWrite 47

Device drivers 9, 21, 43–49
- defaults for 48
- function table for 48
- integrating your own 48

E

End-Of-File (EOF) 14

F

FAT file system 7
File system layer, of *µC/FS* 20
File system layer, of µC/FS 8
FS_ClearErr 34
FS_CloseDir 37
fs_conf.h file 15, 17–21, 17–21, 17–21
FS_Exit 14, 25
FS_FClose 12, 14, 27
FS_FError 12, 14, 35
FS_FOpen 12, 13, 28
FS_FRead 14, 30
FS_FSeek 32
FS_FTell 33
FS_FWrite 12, 31
FS_IDE_HW_X_BusyLedOff 88
FS_IDE_HW_X_BusyLedOn 89
FS_IDE_HW_X_DetectStatus 109
FS_IDE_HW_X_GetAltStatus 91
FS_IDE_HW_X_GetCylHigh 92
FS_IDE_HW_X_GetCylLow 93
FS_IDE_HW_X_GetData 94
FS_IDE_HW_X_GetDevice 95
FS_IDE_HW_X_GetError 96
FS_IDE_HW_X_GetSectorCount 97
FS_IDE_HW_X_GetSectorNo 98
FS_IDE_HW_X_GetStatus 99
FS_IDE_HW_X_HWReset 90
FS_IDE_HW_X_SetCommand 100
FS_IDE_HW_X_SetCylHigh 101
FS_IDE_HW_X_SetCylLow 102
FS_IDE_HW_X_SetData 103
FS_IDE_HW_X_SetDevControl 104
FS_IDE_HW_X_SetDevice 105
FS_IDE_HW_X_SetFeatures 106
FS_IDE_HW_X_SetSectorCount 107
FS_IDE_HW_X_SetSectorNo 108
FS_Init 11, 14, 26
FS_IoCtl 12
FS_MkDir 38
FS_MMC_HW_X_AdjustFOP 76
FS_MMC_HW_X_BusyLedOff 72
FS_MMC_HW_X_BusyLedOn 73
FS_MMC_HW_X_CheckOCR 77
FS_MMC_HW_X_CheckWP 79
FS_MMC_HW_X_ClockCard 74
FS_MMC_HW_X_DetectStatus 80
FS_MMC_HW_X_GetFOP 78
FS_MMC_HW_X_ReadByte 82
FS_MMC_HW_X_ReadByteNoSync 83
FS_MMC_HW_X_ReadSingleBlock 84
FS_MMC_HW_X_SetCS 75
FS_MMC_HW_X_WaitBusy 81
FS_MMC_HW_X_WriteByte 86
FS_MMC_HW_X_WriteByteSingleBlock 87
FS_OpenDir 39
FS_OS_EMBOS 20
FS_OS_TIME_SUPPORT 20
FS_OS_UCOS_II 20
FS_OS_WINDOWS 20
fs_port.h file 15, 21–23
FS_ReadDir 40
FS_Remove 36
FS_RewindDir 41
FS_RmDir 42
FS_SMC_HW_X_BusyLedOff 51
FS_SMC_HW_X_BusyLedOn 52
FS_SMC_HW_X_ChkBusy 59
FS_SMC_HW_X_ChkCardIn 60
FS_SMC_HW_X_ChkPower 61
FS_SMC_HW_X_ChkStatus 62
FS_SMC_HW_X_ChkTimer 68
FS_SMC_HW_X_ChkWP 63

FS_SMC_HW_X_DetectStatus 64
FS_SMC_HW_X_InData 66
FS_SMC_HW_X_OutData 67
FS_SMC_HW_X_SetAddr 53
FS_SMC_HW_X_SetCmd 54
FS_SMC_HW_X_SetData 55
FS_SMC_HW_X_SetStandby 56
FS_SMC_HW_X_SetTimer 69
FS_SMC_HW_X_StopTimer 70
FS_SMC_HW_X_VccOff 57
FS_SMC_HW_X_VccOn 58
FS_SMC_HW_X_WaitTimer 71
FS_USE_FAT_FSL 21
FS_USE_RAMDISK_DRIVER 21
FS_USE_SMC_DRIVER 21
FS_USE_WINDRIVE_DRIVER 21
FS_X_OS_Exit 111
FS_X_OS_GetDate 121
FS_X_OS_GetTime 122
FS_X_OS_Init 112
FS_X_OS_LockDeviceOp 119
FS_X_OS_LockFileHandle 113
FS_X_OS_LockFileOp 117
FS_X_OS_LockMem 114
FS_X_OS_UnlockDeviceOp 120
FS_X_OS_UnlockFileHandle 115
FS_X_OS_UnlockFileOp 118
FS_X_OS_UnlockMem 116
Function table, for device drivers 48

H

Hardware access functions 50–71
control line 51–56
data transfer 66–67
power control 57–58
status detection 59–64
timer 68–71
Hardware access layer 50

I

Include files 15

L

Logical block layer, of µC/FS 9

M

Microsoft compiler 10
MMC device driver 72–110
MultiMedia & SD card device driver ..... *See MMC device driver*

O

OS integration functions 111–22
device access protection 119–20
file access protection 117–18
internal data structure protection 113–16
time/date 121–22
OS support 20

R

RAM disk driver 10, 21

S

Sample code, generic 15
Sample project 10–14
building 10
debugging 10–14
Search path, configuration of 15
SmartMedia card device driver ..... *See SMC device driver*
SMC device driver 21, 50–71
Source code, generic 15

T

Testing µC/FS integration 15
Time/date support 20

W

Windows driver 21