Micriμm

Empowering Embedded Systems

μC/FS

V4.03

User Manual

www.Micrium.com

Disclaimer

Specifications written in this manual are believed to be accurate, but are not guaranteed to be entirely free of error. Specifications in this manual may be changed for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, Micrium assumes no responsibility for any errors or omissions and makes no warranties. Micrium specifically disclaims any implied warranty of fitness for a particular purpose.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of Micrium. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2008-2010 Micriμm, Weston, Florida 33327-1848, U.S.A.

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.

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available. For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: licensing@micrium.com

Contact address

Micrium
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : www.micrium.com
Email : support@micrium.com

Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
V4.00	2009/01/31	BAN	Finalized for release.
V4.01	2009/04/10	BAN	Updated.
V4.02	2009/05/27	BAN	Updated.
V4.03	2010/05/27	FBJ	Updated.

Table Of Contents

I Introduction 12

1 µC/FS Architecture 14
1.01 Architecture Components 16
1.01.01 Your Application 16
1.01.02 LIB (Libraries) 16
1.01.03 POSIX API Layer 16
1.01.04 FS Layer 16
1.01.05 File System Driver Layer 17
1.01.06 Device Driver Layer 17
1.01.07 CPU Layer 17
1.01.08 RTOS Layer 17
1.02 Directories 18
1.02.01 µC/FS Directories 18
1.02.02 µC/FS Template Directories 19
1.02.03 Support Directories 20
1.03 Block Diagram 22

2 Miscellaneous 23
2.01 Nomenclature 23
2.02 µC/FS Device and Volume Names 24
2.03 µC/FS File and Directory Names and Paths 25
2.04 µC/FS Name Lengths 26
2.05 Resource Usage 27

3 Devices and Volumes 28
3.01 Device Operations 29
3.02 Using Devices 30
3.03 Using Removable Devices 32
3.04 Partitions 33
3.05 Volume Operations 36
3.06 Using Volumes 37
3.07 Using Volume Cache 38
3.07.01 Choosing Cache Parameters 39
3.07.02 Other Caching & Buffering Mechanisms 40

4 POSIX API 41
4.01 Supported Functions 42
4.02 Working Directory Functions 43
4.03 File Access Functions 44
4.03.01 Opening, Reading & Writing Files 46
4.03.02 Getting or Setting the File Position 48
4.03.03 Configuring a File Buffer 49
4.03.04 Diagnosing a File Error 50
4.03.05 Atomic File Operations Using File Lock 50
4.04 Directory Access Functions 51
4.05 Entry Access Functions 53

5 Files 54
5.01 File Access Functions 55
5.01.01 Opening Files 56
5.01.02 Getting Information About a File 57
5.01.03 Configuring a File Buffer 58
5.01.04 File Error Functions 58
5.01.05 Atomic File Operations Using File Lock 59
5.02 Entry Access Functions 60
5.02.01 File and Directory Attributes 61
5.02.02 File and Directory Times 62
5.02.03 Creating New Files and Directories 63
5.02.04 Deleting Files and Directories 64

6 Directories 65
6.01 Directory Access Functions 66

7 File Systems: FAT 67
7.01 FAT Architecture 68
7.01.01 FAT12 / FAT16 / FAT32 68
7.01.02 Short and Long File Names 69
7.01.03 Directories and Directory Entries 69
7.01.04 FAT System Driver Architecture 71
7.02 Operations 72
7.02.01 Formatting 72
7.02.02 Disk Check 73
7.02.03 Journaling 74

8 Device Drivers 75
8.01 Provided Device Drivers 76
8.01.01 Driver Characterization 76
8.02 Writing a Device Driver 77
8.02.01 Get Driver Name: FSDev_####_NameGet() 79
8.02.02 Initialize the Driver: FSDev_####_Init() 80
8.02.03 Open (Initialize) a Device Instance: FSDev_####_Open() 81
8.02.04 Close (Uninitialize) a Device Instance: FSDev_####_Close() 82
8.02.05 Read from a Device: FSDev_####_Rd() 83
8.02.06 Write to a Device: FSDev_####_Wr() 84
8.02.07 Get Information about a Device: FSDev_####_Query() 85
8.02.08 Perform I/O Control Operation: FSDev_####_IO_Ctrl() 86
8.03 Writing a Driver SPI BSP 88
8.03.01 SPI Communication 89
8.03.02 Open (Initialize) SPI: `FSDev_####_BSP_SPI_Open()` 90

8.03.03 Close (uninitialize) SPI: FSDev_####_BSP_SPI_Close() 91
8.03.04 Acquire/Release SPI Lock: FSDev_####_BSP_SPI_Lock()/Unlock() 92
8.03.05 Read Data from SD/MMC SPI: FSDev_####_BSP_SPI_Rd() 93
8.03.06 Write Data to SD/MMC SPI: FSDev_####_BSP_SPI_Wr() 94
8.03.07 Enable Chip Select: FSDev_####_BSP_SPI_ChipSelEn() 95
8.03.08 Disable Chip Select: FSDev_####_BSP_SPI_ChipSelDis() 96
8.03.09 Set SPI Clock Frequency: FSDev_####_BSP_SPI_SetClkFreq() 97

9 IDE/CF Driver 98
9.01 Files and Directories 99
9.02 Using the IDE/CF Driver 100
9.02.01 ATA (True IDE) Communication 103
9.02.02 IDE BSP Overview 105
9.02.02.01 Open (Initialize) IDE Device: FSDev_IDE_BSP_Open() 107
9.02.02.02 Close (Uninitialize) IDE Device: FSDev_IDE_BSP_Close() 108
9.02.02.03 Acquire / Release IDE Bus Lock: FSDev_IDE_BSP_Lock()/Unlock() 109
9.02.02.04 Reset IDE Device: FSDev_IDE_BSP_Reset() 110
9.02.02.05 Read Register on IDE Device: FSDev_IDE_BSP_RegRd() 111
9.02.02.06 Write Register on IDE Device: FSDev_IDE_BSP_RegWr() 112
9.02.02.07 Write Command to IDE Device Registers: FSDev_IDE_BSP_CmdWr() 113
9.02.02.08 Read Data from IDE Device: FSDev_IDE_BSP_DataRd() 114
9.02.02.09 Write Data to IDE Device: FSDev_IDE_BSP_DataWr() 115
9.02.02.10 Setup DMA: FSDev_IDE_BSP_DMA_Start() 116
9.02.02.11 End DMA Transfer: FSDev_IDE_BSP_DMA_End() 117
9.02.02.12 Get IDE Drive Number: FSDev_IDE_BSP_GetDrvNbr() 118
9.02.02.13 Get Supported Transfer Modes: FSDev_IDE_BSP_GetModesSupported() 119
9.02.02.14 Set Transfer Mode: FSDev_IDE_BSP_SetMode() 120
9.02.02.15 Delay 400 ns: FSDev_IDE_BSP_Dly400_ns() 121

10 Logical Device Driver 122

11 MSC Driver 123
11.01 Files and Directories 124
11.02 Using the MSC Driver 125

12 NAND Flash Driver 127

13 NOR Flash Driver 128
13.01 Files and Directories 129
13.02 Driver & Device Characteristics 130
13.03 Using a Parallel NOR Device 131
13.03.01 Driver Architecture 134
13.03.02 Hardware 134
13.03.03 NOR BSP Overview 136
13.03.03.01 Open (Initialize) Bus for NOR: FSDev_NOR_BSP_Open() 137
13.03.03.02 Close (Uninitialize) Bus for NOR: FSDev_NOR_BSP_Close() 138
13.03.03.03 Read from Bus Interface: FSDev_NOR_BSP_Rd_XX() 139
13.03.03.04 Read Word from Bus Interface: FSDev_NOR_BSP_RdWord_XX() 140
13.03.03.05 Write Word to Bus Interface: FSDev_NOR_BSP_WrWord_XX() 141
13.03.03.06 Wait while NOR is Busy: FSDev_NOR_BSP_WaitWhileBusy() 142
13.04 Using a Serial NOR Device 143
13.04.01 Hardware 144
13.04.02 NOR SPI BSP Overview 144
13.05 Physical-Layer Drivers 145

13.05.01 FSDev_NOR_AMD_1x08, FSDev_NOR_AMD_1x16 145
13.05.02 FSDev_NOR_Intel_1x16 145
13.05.03 FSDev_NOR_SST39 146
13.05.04 FSDev_NOR_STM25 146
13.05.05 FSDev_NOR_SST25 146

14 RAM Disk Driver 148
14.01 Files and Directories 149
14.02 Using the RAM Disk Driver 150

15 SD/MMC Drivers 153
15.01 Files and Directories 155
15.02 Using the SD/MMC CardMode Driver 156
15.02.01 SD/MMC CardMode Communication 158
15.02.02 SD/MMC CardMode Communication Debugging 160
15.02.03 SD/MMC CardMode BSP Overview 165
15.02.03.01 Open (Initialize) Card Interface: FSDev_SD_Card_BSP_Open() 167
15.02.03.02 Close (Uninitialize) Card Interface: FSDev_SD_Card_BSP_Close() 168
15.02.03.03 Acquire / Release SPI Lock: FSDev_SD_SPI_BSP_Lock()/Unlock() 169
15.02.03.04 Start a Command: FSDev_SD_Card_BSP_CmdStart() 170
15.02.03.05 Wait for Command to End: FSDev_SD_Card_BSP_CmdWaitEnd() 172
15.02.03.06 Read Data following Command: FSDev_SD_Card_BSP_CmdDataRd() 173
15.02.03.07 Write Data following Command: FSDev_SD_Card_BSP_CmdDataWr() 174
15.02.03.08 Get Maximum Block Count: FSDev_SD_Card_BSP_GetBlkCntMax() 175
15.02.03.09 Get Maximum Bus Width: FSDev_SD_Card_BSP_GetBusWidthMax() 176
15.02.03.10 Set Bus Width: FSDev_SD_Card_BSP_SetBusWidth() 177
15.02.03.11 Set Clock Frequency: FSDev_SD_Card_BSP_SetClkFreq() 178
15.02.03.12 Set Data Timeout: FSDev_SD_Card_BSP_SetDataTimeout() 179
15.02.03.13 Set Response Timeout: FSDev_SD_Card_BSP_SetRespTimeout() 180
15.03 Using the SD/MMC SPI Driver 181
15.03.01 SD/MMC SPI Communication 183
15.03.02 SD/MMC SPI Communication Debugging 184
15.03.03 SD/MMC SPI BSP Overview 187

A µC/FS API Reference Manual 188
A.01 General File System Functions 189
FS_DevDrvAdd() 190
FS_Init() 191
FS_VersionGet() 192
FS_WorkingDirGet() 193
FS_WorkingDirSet() 194
A.02 POSIX API Functions 195
fs_asctime_r() 197
fs_chdir() 198
fs_clearerr() 199
fs_closedir() 200
fs_ctime_r() 201
fs_fclose() 202
fs_feof() 203
fs_ferror() 204
fs_fflush() 205
fs_fgetpos() 206
fs_flockfile() 207
fs_fopen() 208

fs_fread() 209
fs_fseek() 210
fs_fsetpos() 211
fs_ftell() 212
fs_ftruncate() 213
fs_ftrylockfile() 214
fs_funlockfile() 215
fs_fwrite() 216
fs_getcwd() 217
fs_localtime_r() 218
fs_mkdir() 219
fs_mktime() 220
fs_opendir() 221
fs_readdir_r() 222
fs_remove() 223
fs_rename() 225
fs_rewind() 227
fs_rmdir() 228
fs_setbuf() 229
fs_setvbuf() 230
A.03 Device Functions 231
FSDev_Close() 233
FSDev_GetDevName() 234
FSDev_GetDevCnt() 235
FSDev_GetDevCntMax() 236
FSDev_GetNbrPartitions() 237
FSDev_Open() 238
FSDev_PartitionAdd() 240
FSDev_PartitionFind() 241
FSDev_PartitionInit() 242
FSDev_Query() 243
FSDev_Rd() 244
FSDev_Refresh() 245
FSDev_Wr() 246
A.04 Directory Access Functions 247
FSDir_Close() 248
FSDir_Open() 249
FSDir_Rd() 250
A.05 Entry Access Functions 251
FSEntry_AttribSet() 252
FSEntry_Copy() 253
FSEntry_Create() 254
FSEntry_Del() 256
FSEntry_Query() 257
FSEntry_Rename() 258
FSEntry_TimeSet() 260
A.06 File Functions 261
FSFile_BufAssign() 263
FSFile_BufFlush() 264
FSFile_Close() 265
FSFile_ClrErr() 266
FSFile_IsEOF() 267
FSFile_IsErr() 268
FSFile_LockAccept() 269
FSFile_LockGet() 270
FSFile_LockSet() 271

FSFile_Open() 272
FSFile_PosGet() 274
FSFile_PosSet() 275
FSFile_Query() 276
FSFile_Rd() 277
FSFile_Truncate() 278
FSFile_Wr() 279
A.07 Time Functions 280
FSTime_TimeGet() 281
FSTime_TimeSet() 282
FSTime_Time_to_Str() 283
FSTime_Time_to_TS() 284
FSTime_TS_to_Str() 285
FSTime_TS_to_Time() 286
A.08 Volume Functions 287
FSVol_Close() 289
FSVol_Fmt() 290
FSVol_GetDfltVolName() 291
FSVol_GetVolCnt() 292
FSVol_GetVolCntMax() 293
FSVol_GetVolName() 294
FSVol_IsDflt() 295
FSVol_IsMounted() 296
FSVol_LabelGet() 297
FSVol_LabelSet() 298
FSVol_Open() 299
FSVol_Query() 301
FSVol_Rd() 302
FSVol_Wr() 303
A.09 Volume Cache Functions 304
FSVol_CacheAssign () 305
FSVol_CacheInvalidate () 307
FSVol_CacheFlush () 308
A.10 NOR Driver Functions 309
FSDev_NOR_LowFmt() 310
FSDev_NOR_LowMount() 311
FSDev_NOR_LowUnmount() 312
FSDev_NOR_LowCompact() 313
FSDev_NOR_LowDefrag() 314
FSDev_NOR_PhyRd() 315
FSDev_NOR_PhyWr() 316
FSDev_NOR_PhyEraseBlk() 317
FSDev_NOR_PhyEraseChip() 318
A.11 SD/MMC Driver Functions 319
FSDev_SD_xxx_QuerySD() 320
FSDev_SD_xxx_RdCID() 321
FSDev_SD_xxx_RdCSD() 322
A.12 FAT System Driver Functions 323
FS_FAT_JournalOpen() 324
FS_FAT_JournalClose() 325
FS_FAT_JournalStart() 326
FS_FAT_JournalStop() 327
FS_FAT_VolChk() 328
A.13 BSP Functions 329
FS_BSP_TimeGet() 330
FS_BSP_TimeSet() 331

B µC/FS Error Codes 332
B.01 System Error Codes 332
B.02 Buffer Error Codes 332
B.03 Cache Error Codes 332
B.04 Device Error Codes 333
B.05 Device Driver Error Codes 333
B.06 Directory Error Codes 333
B.07 ECC Error Codes 333
B.08 Entry Error Codes 334
B.09 File Error Codes 334
B.10 Name Error Codes 334
B.11 Partition Error Codes 335
B.12 Pools Error Codes 335
B.13 File System Error Codes 335
B.14 Volume Error Codes 335

C µC/FS Types and Structures 337
FS_CFG 338
FS_DATE_TIME 339
FS_DEV_INFO 340
FS_DEV_NOR_CFG 341
FS_DEV_RAM_CFG 343
FS_DIR_ENTRY (struct dirent) 344
FS_ENTRY_INFO 345
FS_FAT_SYS_CFG 346
FS_PARTITION_ENTRY 347
FS_TS 348
FS_VOL_INFO 349

D µC/FS Configuration 350
D.01 File System Configuration 351
FS_CFG_SYS_DRV_SEL 351
FS_CFG_CACHE_EN 351
FS_CFG_API_EN 351
FS_CFG_DIR_EN 351
D.02 Feature Inclusion Configuration 352
FS_CFG_FILE_BUF_EN 352
FS_CFG_FILE_LOCK_EN 352
FS_CFG_PARTITION_EN 352
FS_CFG_WORKING_DIR_EN 353
FS_CFG_UTF8_EN 353
FS_CFG_RD_ONLY_EN 353
D.03 Name Restriction Configuration 354
FS_CFG_MAX_PATH_NAME_LEN 354
FS_CFG_MAX_FILE_NAME_LEN 354
FS_CFG_MAX_DEV_DRV_NAME_LEN 354
FS_CFG_MAX_DEV_NAME_LEN 354
FS_CFG_MAX_VOL_NAME_LEN 354
D.04 Debug Configuration 355
FS_CFG_DBG_MEM_CLR_EN 355
FS_CFG_DBG_WR_VERIFY_EN 355
D.05 Argument Checking Configuration 356
FS_CFG_ARG_CHK_EXT_EN 356

FS_CFG_ARG_CHK_DBG_EN 356
D.06 File System Counter Configuration 357
FS_CFG_CTR_STAT_EN 357
FS_CFG_CTR_ERR_EN 357
D.07 FAT Configuration 358
FS_FAT_CFG_LFN_EN 358
FS_FAT_CFG_FAT12_EN 358
FS_FAT_CFG_FAT16_EN 358
FS_FAT_CFG_FAT32_EN 358
FS_FAT_CFG_JOURNAL_EN 358
FS_FAT_CFG_VOL_CHK_EN 358
FS_FAT_CFG_VOL_CHK_MAX_LEVELS 358
D.08 SD/MMC SPI Configuration 359
FS_DEV_SD_SPI_CFG_CRC_EN 359
D.09 Tinestamp Configuration 360
FS_CFG_GET_TS_FROM_OS 360
D.10 Trace Configuration 361

E Shell Commands 362
E.01 Files and Directories 363
E.02 Using the Shell Commands 364
E.03 Commands 366
fs_cat 367
fs_cd 368
fs_cp 369
fs_date 370
fs_df 371
fs_ls 372
fs_mkdir 373
fs_mkfs 374
fs_mount 375
fs_mv 376
fs_od 377
fs_pwd 378
fs_rm 379
fs_rmdir 380
fs_touch 381
fs_umount 382
fs_wc 383
E.04 Configuration 384
FS_SHELL_CFG_BUF_LEN 384
FS_SHELL_CFG_CMD_####_EN 384

F References 385

Introduction

µC/FS is a compact, reliable, high-performance file system.

The source code for µC/FS consists of clean, consistent ANSI C source code. C was chosen because C is still the most predominant language in the embedded industry. Over 50% of the code actually consists of comments. Most global variables and all functions are described. References to appropriate documents are referenced in the code when applicable. The code is designed to be used with just about any CPU, RTOS and memory device.

I.01 Portable

µC/FS was designed for resource constrained embedded applications. Although µC/FS can work on some 8- and 16-bit processors, µC/FS will work best with 32- or 64-bit CPUs.

I.02 Scalable

The memory footprint of µC/FS can be adjusted at compile time based on the features you need and the desired level of run-time argument checking. For applications with limited RAM, features such as cache and read/write buffering can be disabled; for applications with the sufficient RAM, these features can be enabled in order to gain better performance.

I.03 Coding Standards

Coding standards have been established early in the design of µC/FS and include the following:

- C coding style
- Naming conventions for `#define` constants, macros, variables and functions
- Commenting
- Directory structure

.

I.04 MISRA C

The source code for µC/FS follows almost all MISRA (the Motor Industry Software Reliability Association) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA consortium include Delco Electronics, Ford

Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and other firms and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site, http://www.misra.org.uk.

I.05 RTOS

µC/FS does not assume the presence of an RTOS. However, if you are using an RTOS, a simple port layer is required (consisting of a few semaphores), in order to prevent simultaneous access to core structures from different tasks. If you are not using an RTOS, then this port layer may consist of empty functions.

I.06 Device Drivers

Device drivers are available for most common media, including

- SD/MMC cards
- NOR flash
- Serial NOR flash
- RAM disk
- IDE/CF

Note that a device driver for NAND flash should be released soon.

Each of these is written with a layered structure so that it can easily be ported to your hardware. The device driver structure is simple—basically just initialization read and write functions—so that it can be easily ported to a new medium.

µC/FS can access multiple media simultaneously, including multiple instances of the same type of medium (since all drivers are re-entrant).

I.07 Application Interface

µC/FS provides two APIs for file and directory access:

1. A standard POSIX-compatible API is provided, including functions like `fs_fwrite()`, `fs_fread()`, `fs_fsetpos()`, etc., that offer the same arguments and return values as the POSIX functions `fwrite()`, `fread()`, `fsetpos()`, etc.

2. A clean, consistent API with parallel argument placement and meaningful return error codes. This API has functions such as `FSFile_Wr()`, `FSFile_Rd()`, `FSFile_PosSet()`, etc.

Chapter

1

µC/FS Architecture

µC/FS was written from the ground up to be modular and easy to adapt to different CPUs (Central Processing Units), RTOSs (Real-Time Operating Systems), file system media and compilers. Figure 1-1 shows a simplified block diagram of the different µC/FS modules and their relationships.

Notice that all of the µC/FS files start with '`fs_`'. This convention allows you to quickly identify which files belong to µC/FS. Also note that all functions and global variables start with '`FS`', and all macros and `#defines` start with '`FS_`'.

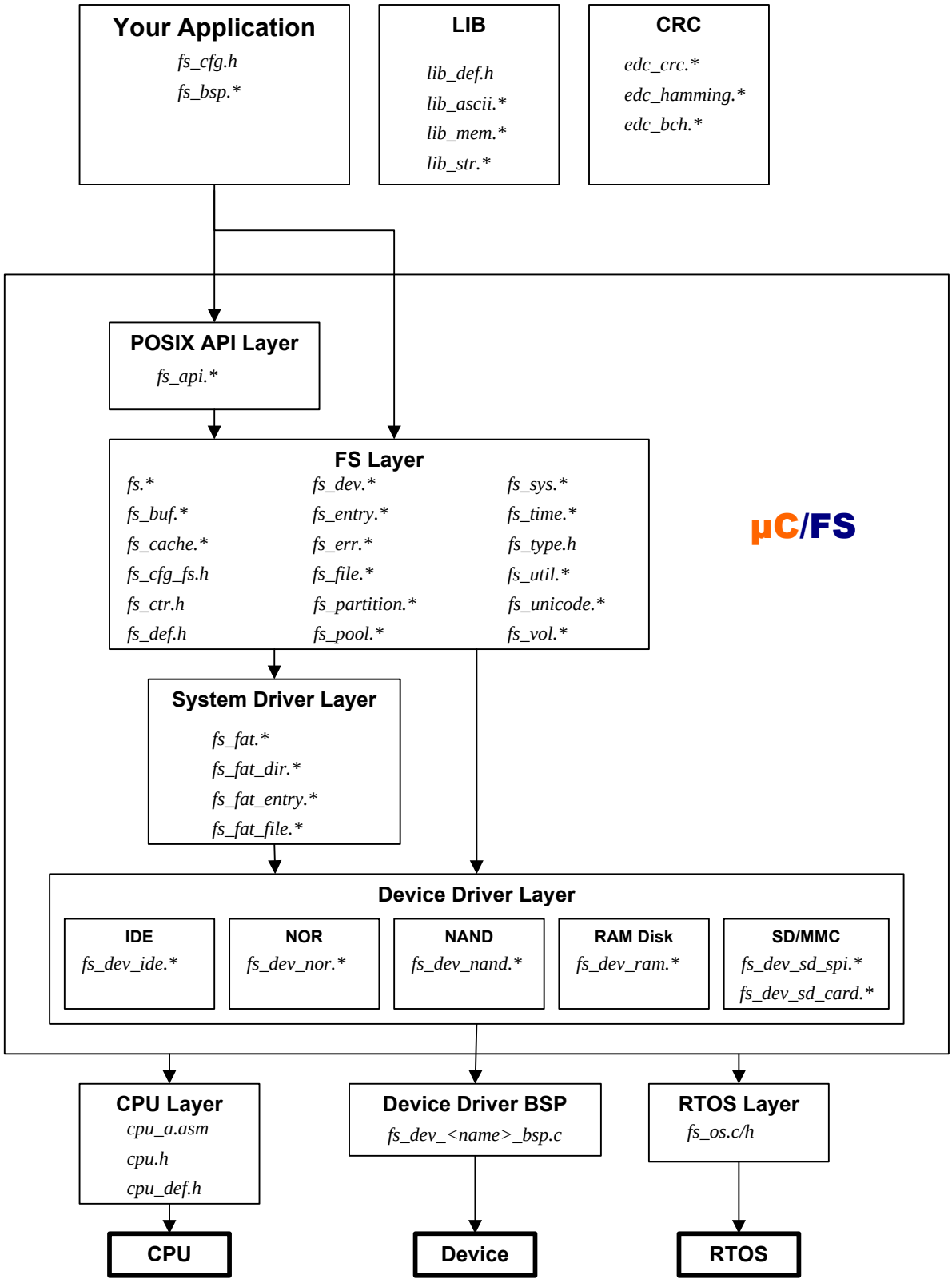


Figure 1-1. µC/FS architecture.

1.01 Architecture Components

µC/FS consists of a set of modular software components. It also requires a few external components (provided with the release) be compiled into the application and a few configuration and BSP files be adapted to the application.

1.01.01 Your Application

Your application needs to provide configuration information to µC/FS in the form of one C header file named *fs_cfg.h*.

Some of the configuration data in *fs_cfg.h* consist of specifying whether certain features will be present. For example, LFN support, volume cache and file buffering are all enabled or disabled in this file. In all, there are about 30 `#define` to set. However, most of these can be set to their default values.

1.01.02 LIB (Libraries)

Because µC/FS is designed to be used in safety critical applications, all 'standard' library functions like `strcpy()`, `memset()`, etc., have been re-written to follow the same quality as the rest of the file system software.

1.01.03 POSIX API Layer

Your application interfaces to µC/FS using the well-known *stdio.h* API (Application Programming Interface). Alternately, you can use µC/FS's own file and directory interface functions. Basically, POSIX API layer is a layer of software that converts POSIX file access calls to µC/FS file access calls.

1.01.04 FS Layer

This layer contains most of the CPU-, RTOS- and compiler-independent code for µC/FS. There are three categories of files in this section:

1. **File system object-specific files:**
    a. Devices (*fs_dev.**)
    b. Directories (*fs_dir.**)
    c. Entries (*fs_entry.**)
    d. Files (*fs_file.**)
    e. Volumes (*fs_vol.**)

2. **Support files:**
    a. Buffer management (*fs_buf.**)
    b. Cache management (*fs_cache.**)
    c. Counter management (*fs_ctr.h*)
    d. Pool management (*fs_pool.**)
    e. File system drivers (*fs_sys.**)
    f. Time functions (*fs_time.**)
    g. Utility functions (*fs_util.**)

3. **Miscellaneous header files:**
    a. Master µC/FS header file (*fs.h*)
    b. Error codes (*fs_err.h*)
    c. Miscellaneous data types (*fs_type.h*)
    d. Miscellaneous definitions (*fs_def.h*)

e. Configuration definitions (*fs_cfg_fs.h*)

1.01.05 File System Driver Layer

The file system driver layer understands the organization of a particular file system type, such as FAT. The current version of µC/FS only supports FAT file systems. ***fs_fat*.*** contains the file system driver which should be used for FAT12/FAT16/FAT32 disks with or without Long File Name (LFN) support.

1.01.06 Device Driver Layer

The device driver layer understands about types of file system media (SD/MMC card, NOR flash, etc.). In order for the device drivers to be independent of your CPU, we use additional files to encapsulate such details as the access of registers, reading and writing to a data bus and setting clock rates.

Each device driver is named according to the pattern

fs_dev_<dev drv name>.c

where <*dev drv name*> is the an identifier for the device driver. For example, the driver for SD/MMC cards using SPI mode is called *fs_dev_sd_spi*.c. Most device drivers require a BSP layer, with code for accessing registers, reading from or writing to a data bus, etc. This file is named according to the pattern

fs_dev_<dev drv name>_bsp.c

For example, *fs_dev_sd_spi_bsp.c* contains the BSP functions for the driver SD/MMC cards using SPI mode.

1.01.07 CPU Layer

µC/FS can work with either an 8, 16, 32 or even 64-bit CPU, but needs to have information about the CPU you are using. The CPU layer defines such things as the C data type corresponding to 16-bit and 32-bit variables, whether the CPU is little- or big-endian and, how interrupts are disabled and enabled on the CPU, etc.

CPU specific files are found in the ***…\uC-CPU*** directory and, in order to adapt µC/FS to a different CPU, you would need to either modify the *cpu*.** files or, create new ones based on the ones supplied in the ***uC-CPU*** directory. In general, it's much easier to modify existing files because you have a better chance of not forgetting anything.

1.01.08 RTOS Layer

µC/FS does not require an RTOS. However, if µC/FS is used with an RTOS, a set of functions must be implemented to prevent simultaneous access of devices and core µC/FS structures by multiple tasks.

µC/FS is provided with a no-RTOS interface (which contains just empty functions) and a µC/OS-II interface. If you use a different RTOS, you can use the *fs_os.** for µC/OS-II as a template to interface to the RTOS of your choice.

1.02 Directories

The files shown in Figure 1-1 and Figure 1-2 are placed in a directory structure such as to group common elements. The µC/FS distribution contains the following directories.

1.02.01 µC/FS Directories

\Micrium\Software\uC-FS
This is the main directory for µC/FS.

\Micrium\Software\uC-FS\Dev
This directory contains device-specific files. Currently, µC/FS supports several device types:

\Micrium\Software\uC-FS\Dev\IDE
This directory contains the IDE driver and related files. For more information, see Chapter 9.

\Micrium\Software\uC-FS\Dev\MSC
This directory contains the MSC driver (for use with µC/USB-Host) and related files. For more information, see Chapter 11.

\Micrium\Software\uC-FS\Dev\NOR
This directory contains the NOR driver and related files. For more information, see Chapter 13.

\Micrium\Software\uC-FS\Dev\RAM
This directory contains the RAM disk driver and related files. For more information, see Chapter 14.

\Micrium\Software\uC-FS\Dev\SD
This directory contains the SD/MMC drivers and related files. For more information, see Chapter 15.

\Micrium\Software\uC-FS\FAT
This directory contains the FAT file system driver. These files are necessary if using a FAT12, FAT16 or FAT32 file system, with or without Long File Name (LFN) support.

\Micrium\Software\uC-FS\OS
This directory contains the RTOS abstraction layer which allows you to use µC/FS with just about any commercial or in-house RTOS, or without any RTOS at all. You would place the abstraction layer for your RTOS in a subdirectory under OS as follows:

\Micrium\Software\uC-FS\OS\rtos_name\fs_os.*

Note that you must always name the files for your RTOS abstraction layer *fs_os.**.

µC/FS has been tested both without an RTOS and with µC/OS-II or µC/OS-III. The RTOS layer files which should be used without an RTOS are found in

\Micrium\Software\uC-FS\OS\None

The RTOS layer files for µC/OS-II are found in

\Micrium\Software\uC-FS\OS\uCOS-II

The RTOS layer files for µC/OS-III are found in

\Micrium\Software\uC-FS\OS\uCOS-III

\Micrium\Software\uC-FS\Source

This directory contains all the CPU-, medium- and RTOS-independent files as shown in Figure 1-1. You should not have to change anything in this directory in order to use µC/FS.

\Micrium\Software\µC-FS\Doc

This directory contains µC/FS documentation files including this document.

\Micrium\Software\µC-FS\Examples

This directory contains a few bsp driver examples. These examples are provided as is in order to help you develop the low level, platform dependent, drivers.

The examples under this directory are provided with no garantee. Technical support in regards of those examples are not covered by your regular maintenance agreement. Should you need any help in getting those examples working on your platform, please contact our sales department.

1.02.02 µC/FS Template Directories

µC/FS assumes the presence of a configuration file, plus one or more BSP files, which adapt the file system software or one of its device drivers to your hardware. For each of these, a template file is provided which should be copied to your application directory and modified (if necessary). The following template files are provided:

\Micrium\Software\uC-FS\APP\Template\fs_app.*

These files provide a template for initializing the file system.

\Micrium\Software\uC-FS\BSP\Template\fs_bsp.*

Several application-level functions are required by µC/FS to provide information for the file system. For example, *FS_BSP_GetTime()* should be modified to use a Real Time Clock (RTC) or OS function to get the current date and time to place in the file system metadata for newly created or updated files. Most of these functions (including *FS_BSP_GetTime()*) do not need to be implemented, unless you want the additional functionality provided.

\Micrium\Software\uC-FS\CFG\Template\fs_cfg.h

A configuration file is used to configure the availability of certain features, the quantity of available file system objects, and other software capabilities. Most of the 30+ `#define` in this file will not need to be changed; however, it should be copied to your application directory, in case you will need to make modifications.

\Micrium\Software\uC-FS\Dev

In order for the device drivers to be independent of your CPU, we use additional files to encapsulate such details as the access of registers, reading and writing to a data bus and setting clock rates. Typically, template BSP files are located in directories named like

\Micrium\Software\uC-FS\Dev\<dev drv>\Template\fs_<dev drv name>_bsp.c

1.02.03 Support Directories

µC/FS assumes the presence of a number of support files as described in this section.

\Micrium\Software\uC-CPU

This directory contains files that are used to adapt to different CPUs/compilers. Micrium provides these files for all internally supported architectures.

cpu_def.h

This file contains definitions used by the other CPU specific files. In fact, cpu_def.h should be independent of the actual CPU used.

Within the *uC-CPU* directory, we define processor specific files.

\Micrium\Software\uC-CPU\<cpu_type>\<compiler>

cpu.h

This file contains definitions of data word sizes. Specifically, you define here what C data types are associated with 8, 16 and 32 bit integers and, 32 and 64 bit floating point numbers. *cpu.h* also defines endianness, critical section macros `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`, and more.

cpu_a.s

This file contains functions to implement critical section protection for the specific CPU type used.

\Micrium\Software\uC-LIB

This directory contains library support files that are independent of the CPU, µC/FS and compiler. Micriµm doesn't uses standard C library functions in order to simplify third-party certification.

lib_def.h

This file contains definitions that can be used by other applications. All #defines in this file are prefixed by `DEF_` and contains definitions for `DEF_TRUE` and `DEF_FALSE`, `DEF_YES` and `DEF_NO`, `DEF_ON` and `DEF_OFF` and bit masks.

lib_mem.*

These files contain functions to copy memory, clear memory, etc. All functions in this file start with `Mem_`. Library memory management functions are required for µC/FS and above.

lib_str.*

These files contain function to copy strings, get length of strings, etc..

lib_ascii.*

These files contain functions and macros to classify characters and map character case.

\Micrium\Software\uC-CRC

This sub-directory contains libraries that are used for calculating error correcting codes (ECCs), such as Hamming codes, and error detecting codes (EDCs), such as CRCs. The files in this directory are only needed if you use certain drivers.

edc_crc.*

These files contain code to calculate cyclic redundancy checksums (CRCs).

ecc_hamming.*

These files contain code to calculate Hamming codes. Hamming codes can correct a 1-bit error and detect a 2-bit error in a data buffer.

ecc_bch.*

These files contain code to calculate BCH codes. Codes for correcting 4-bit and 8-bit errors are included.

1.03 Block Diagram

Figure 1-2 shows a block diagram of the modules found in µC/FS and their relationship. Also included are the names of the files that are related to µC/FS.

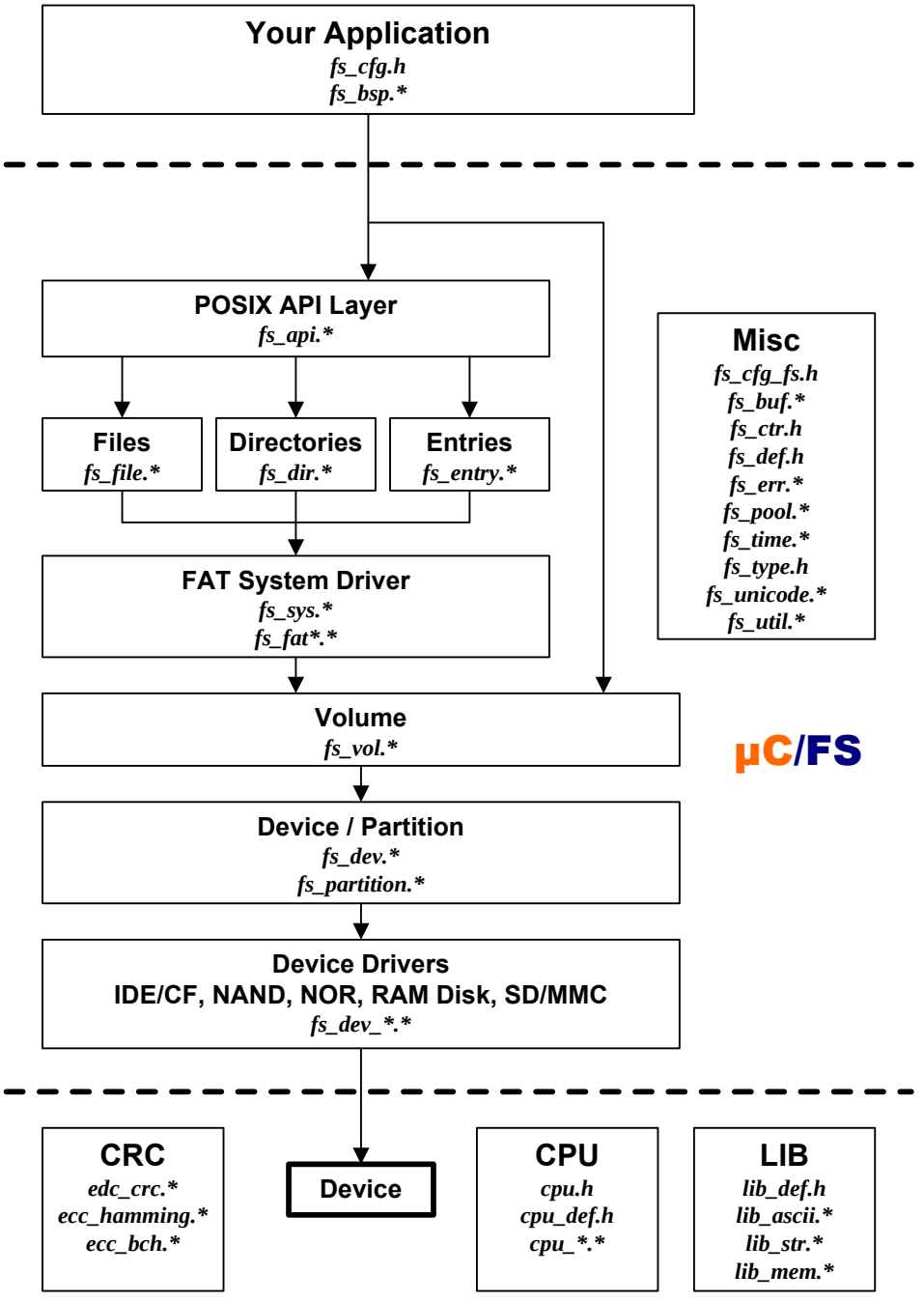


Figure 1-2. µC/FS block diagram.

Chapter

2

Miscellaneous

This chapter provides information on various concepts used in µC/FS. We decided to include this chapter early in the µC/FS manual so that you can start using µC/FS as soon as possible. In fact, we assume you know little about µC/FS and file systems. Concepts will be introduced as needed.

2.01 Nomenclature

This manual uses a set of terms to consistently describe operation of µC/FS and its hardware and software environment. The following is a small list of these terms, with definitions.

A **file system suite** is software which can find and access files and directories. Using "file system suite" rather than "file system" eliminates any need for disambiguation among the second term's several meanings, which include "a system for organizing directories and files", "a collection of files and directories stored on a drive" and (commonly) the software which will be referred to as a file system suite. The term **file system** will always mean a collection of files and directories stored on a drive (or, in this document, volume).

A **device driver** (or just **driver**) is a code module which allows the general-purpose file system suite to access a specific type of device. A device driver is **registered** with the file system suite.

A **device** is an instance of a device type that is accessed using a device driver. An addressable area (typically of 512 bytes) on a device is a **sector**. A sector is the smallest area that (from the file system suite's point of view) can be atomically read or written.

Several devices can use the same device driver. These are distinguished by each having a unique **unit number**. Consequently, **`<DEVICE NAME>:<UNIT NUMBER>:`** is a unique device identifier if all devices are required to have unique names. That requirement is enforced in this file system suite.

A **logical device** is the combination of two or more separate devices. To form a logical device, the sector address spaces of the constituent devices are concatenated to form a single continuous address space.

A device can be **partitioned**, or subdivided into one or more regions (called **partitions**) each consisting of a number of consecutive sectors. Typically, structures are written to the device instructing software as to the location and size of these partitions. This file system suite supports **DOS partitions**.

A **volume** is a device or device partition with a file system. A device or device partition must go through a process called **mounting** to become a volume, which includes finding the file system and making it ready for use. The name by which a volume is addressed may also be called the volume's **mount point**.

A device or volume may be **formatted** to create a new file system on the device. For disambiguation purposes, this

process is also referred to as **high-level formatting**. The volume or device will automatically be mounted once formatting completes.

For certain devices, it is either necessary or desirable to perform **low-level formatting**. This is the process of associating logical sector numbers with areas of the device.

A **file system driver** is a code module which allows the general-purpose file system suite to access a specific type of file system. For example, this file system suite includes a FAT file system driver.

FAT (File Allocation Table) is a common file system type, prevalent in removable media that must work with various OSs. It is named after its primary data structure, a large table that records what clusters of the disk are allocated. A **cluster**, or group of sectors, is the minimum data allocation unit of the FAT file system.

2.02 µC/FS Device and Volume Names

Devices are specified by name. For example, a device can be opened:

```
FSDev_Open("sd:0:", (void *)0, &err);
```

In this case, "sd:0:" is the device name. It is a concatenation of

`sd`	The name of the device driver
`:`	A single colon
`0`	The unit number
`:`	A final colon

The unit number allows multiple devices of the same type; for example, there could be several SD/MMC devices connected to the CPU: "sd:0:", "sd:1", "sd:2", …
The maximum length of a device name is `FS_CFG_MAX_DEV_NAME_LEN`; this must be at least three characters larger than the maximum length of a device driver name, `FS_CFG_MAX_DEV_DRV_NAME_LEN`. A device name (or device driver name) must not contain the characters:

```
:     \     /
```

Volumes are also specified by name. For example, a volume can be formatted:

```
FSVol_Fmt("vol:", (void *)0, &err);
```

In this case, "vol:" is the volume name. µC/FS imposes no special restrictions on these names, except that they must end with a colon ('`:`') , must be no more than `FS_CFG_MAX_VOL_NAME_LEN` characters long, and must not contain either of the characters '`\`' or '`/`':

It is typical to name a volume the same as a device; for example, a volume may be opened:

```
FSVol_Open("sd:0:"          (a)
           "sd:0:"          (b)
           (void *)0,
           &err);
```

In this case, the name of the volume (a) is the same as the name as the device (b). When multiple volumes exist in the same application, the volume name should be prefixed to the file or directory path name:

```
p_file = fs_fopen("sd:0:\\dir01\file01.txt", "w");  // File on SD card
p_file = fs_fopen("ram:0:\\dir01\file01.txt", "w"); // File on RAM disk
```

2.03 µC/FS File and Directory Names and Paths

Files and directories on a volume are identified by a path string; for example, a file can be opened:

```
p_file = fs_fopen("\\test\\file001.txt", "w");
```

In this case, "\\test\\file001.txt" is the path string.

An application specifies the path of a file or directory using either an absolute or a relative path. An **absolute path** is a character string which specifies a unique file, and follows the pattern:

```
<vol_name>:<... Path ...><File>
```

where

`<vol_name>` is the name of the volume, identical to the string specified in `FSVol_Open()`.

`<... Path ...>` is the file path, which MUST always begin AND end with a '\'.

`<File>` is the file (or leaf directory) name, including any extension.

For example:

```
p_file = fs_fopen("sd:0:\\file.txt", "w");           (a)
p_file = fs_fopen("\\file.txt", "w");                (b)
p_file = fs_fopen("sd:0:\\dir01\\file01.txt", "w");  (c)
p_file = fs_opendir("sd:0:\\")                       (d)
p_file = fs_opendir("\\")                            (e)
p_file = fs_opendir("sd:0:\\dir01\\")                (f)
```

Which demonstrate (a) opening a file in the root directory of a specified volume; (b) opening a file in the root directory on a default volume; (c) opening a file in a non-root directory; (d) opening the root directory of a specified volume; (e) opening the root directory of the default volume; (f) opening a non-root directory.

Relative paths can be used if working directories are enabled (`FS_CFG_WORKING_DIR_EN` is `DEF_ENABLED`; see Section D.02.05). A **relative path** begins with neither a volume name nor a '`\`':

```
<... Relative Path ...><File>
```

where

`<... Relative Path ...>` is the file path, which must not begin with a '\' BUT must end with a '\'.

`<File>` is the file (or leaf directory) name, including any extension.

Two special path components can be used. ".." moves the path to the parent directory. "." keeps the path in the same directory (basically, it does nothing).

A relative path is appended to the **current working directory** of the calling task to form the absolute path of the file or directory. The working directory functions, `fs_chdir()` and `fs_getcwd()`, can be used to set and get the working directory.

2.04 µC/FS Name Lengths

The configuration constants FS_CFG_MAX_PATH_NAME_LEN, FS_CFG_MAX_FILE_NAME_LEN and FS_CFG_MAX_VOL_NAME_LEN in *fs_cfg.h* set the maximum length of path names, file names and volume names. The constant FS_CFG_MAX_FULL_NAME_LEN is defined in *fs_cfg_fs.h* to describe the maximum full name length. The **path name** begins with a path separator character and includes the file name; the **file name** is just the portion of the path name after the last (non-final) path separator character. The **full name** is composed of an explicit volume name (optionally) and a path name; the maximum full name length may be calculated:

$$FullNameLen_{max} = VolNameLen_{max} + PathNameLen_{max}$$

Figure 2-3 demonstrates these definitions.

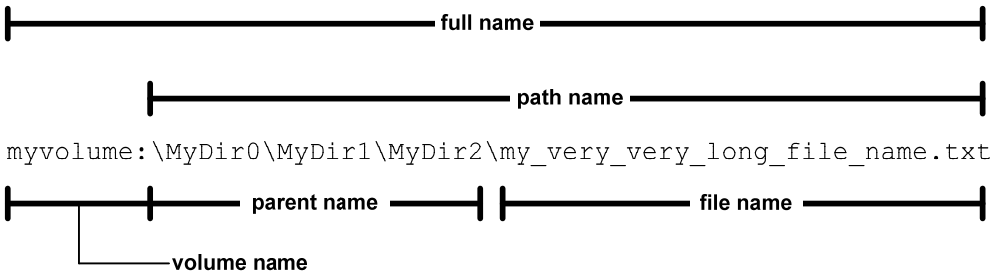


Figure 2-3. File, path and volume name lengths

No maximum parent name length is defined, though one may be derived. The parent name must be short enough so that the path of a file in the directory would be valid. Strictly, the minimum file name length is 1 character, though some OSs may enforce larger values (eleven on some Windows systems), thereby decreasing the maximum parent name length.

$$ParentNameLen_{max} = PathNameLen_{max} - FileNameLen_{min} - 1$$

The configuration constants FS_CFG_MAX_DEV_DRV_NAME_LEN and FS_CFG_MAX_DEV_NAME_LEN in *fs_cfg.h* set the maximum length of device driver names and device names, as shown in Figure 2-4. The device name is between three and five characters longer than the device driver name, since the unit number (the integer between the colons of the device name) must be between 0 and 255.

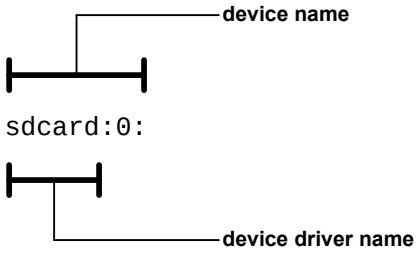


Figure 2-3. Device and device driver name lengths

Each of the maximum name length configurations specifies the maximum string length WITHOUT the NULL character. Consequently, a buffer which holds one of these names must be one character longer than the define value.

2.05 Resource Usage

µC/FS resource usage, of both ROM and RAM, depends heavily on application usage. How many (and which) interface functions are referenced determines the code and constant data space requirements. The greater the quantity of file system objects (buffers, files, directories, devices and volumes) , the more RAM needed.

Table 2-1 give the ROM usage for the file system core, plus additional components that can be included optionally, collected on IAR EWARM v5.4. The 'core' ROM size includes **ALL** file system components and functions (except those itemized in the table); this is significantly larger than most installations because most applications use a fraction of the API.

Component	ROM, Thumb Mode		ROM, ARM Mode	
	High Size Opt	High Speed Opt	High Size Opt	High Speed Opt
Core*	44.1 kB	52.5 kB	66.5 kB	79.4 kB
OS port (µC/OS-III)	0.2 kB	0.2 kB	2.2 kB	2.4 kB
LFN support	6.5 kB	6.7 kB	9.3 kB	9.6 kB
Directories	1.6 kB	2.2 kB	2.4 kB	3.3 kB
Volume check	2.9 kB	3.2 kB	4.7 kB	5.3 kB
Partitions	2.7 kB	3.0 kB	3.7 kB	4.2 kB

Table 2-1. ROM Requirements.
*Includes code and data for ALL file system components and functions except those itemized in the table.

RAM requirements are summarized in Table 2-2. The total depends on the number of each object allocated and the maximum sector size (set by values passed to `FS_Init()` in the file system configuration structure), and various name length configuration parameters (see Section D.03).

Item	RAM (bytes)
Core	360
Per device	56 + FS_CFG_MAX_DEV_NAME_LEN
Per volume	166 + FS_CFG_MAX_VOL_NAME_LEN
Per file	132
Per directory	48
Per buffer	36 + MaxSectorSize
Per device driver	20 bytes
Working directories	(FS_CFG_MAX_PATH_NAME_LEN * 2) * TaskCnt§

Table 2-2. RAM Characteristics
§The number of tasks that use relative path names

See also Section 8.01.01 for ROM/RAM characteristics of file system suite drivers.

Chapter

3

Devices and Volumes

To begin reading files from a medium or creating files on a medium, that medium (hereafter called a device) and the driver which will be used to access it must be registered with the file system. After that, a volume must be opened on that device (analogous to "mounting"). This operation will succeed if and only if the device responds and the file system control structures (for FAT, the Boot Parameter Block or BPB) are located and validated.

In this manual, as in the design of µC/FS, the terms 'device' and 'volume' have distinct, non-overlapping meanings. We define a 'device' as

> A **file system device** is a single physical or logical entity which contains a continuous sequence of addressable sectors. An SD/MMC card is a physical device.

We define a 'volume' as

> A **file system volume** is a collection of files and directories on a device.

These definitions were selected so that multiple volumes could be opened on a device (as shown in Figure 3-1) without requiring ambiguous terminology.

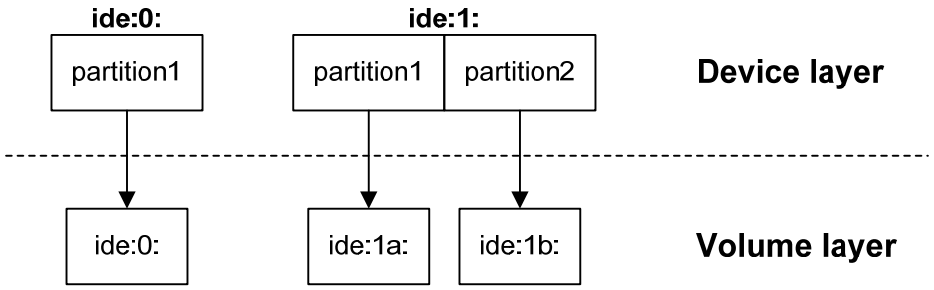


Figure 3-1. Device and volume architecture.

3.01 Device Operations

The ultimate purpose of a file system device is to hold data. Consequently, two major operations that can occur on a device are the reading and writing of individual sectors. Five additional operations can be performed which affect not just individual sectors, but the whole device:

- A device can be **opened**. During the opening of a device, it is initialized and its characteristics are determined (sector size, number of sectors, vendor).
- A device can be **partitioned**. Partitioning divides the final unallocated portion of the device into two parts, so that a volume could be located on each (see Section 3.01.02).
- A device can be **low-level formatted**. Some device must be low-level formatted before being used.
- A device can be **(high-level) formatted**. (High-level) formatting writes the control information for a file system to a device so that a volume on it can be mounted. Essentially, (high-level) formatting is the process of creating a volume on an empty device or partition.
- A device can be **closed**. During the closing of a device, it is uninitialized (if necessary) and associated structures are freed.

These operations and the corresponding API functions are discussed in this section. For information about using device names, see Section 2.04.

Function	Description
`FSDev_Close()`	Remove device from file system.
`FSDev_GetNbrPartitions()`	Get number of paritions on a device.
`FSDev_IO_Ctrl()`	Perform device I/O control operation.
`FSDev_Open()`	Add device to file system.
`FSDev_PartitionAdd()`	Add partition to device.
`FSDev_PartitionFind()`	Find partition on device and get information about partition.
`FSDev_PartitionInit()`	Initialize partition on device.
`FSDev_Query()`	Get device information.
`FSDev_Rd()`	Read sector on device.
`FSDev_Refresh()`	Refresh device in file system.
`FSDev_Wr()`	Write sector on device.

Table 3-1. Device API functions

3.02 Using Devices

A device is opened with `FSDev_Open()`:

```
FSDev_Open((CPU_CHAR *)“ide:0:”,      <-- (a) device name
           (void     *) 0,            <-- (b) pointer to configuration
           (FS_ERR   *)&err);         <-- (c) return error
```

The parameters are the device name (a) and a pointer to a device driver-specific configuration structure (b). If a device driver requires no configuration structure (as the IDE/CF driver does not), the configuration structure (b) should be passed a NULL pointer. For other devices, like RAM disks, this MUST point to a valid structure.

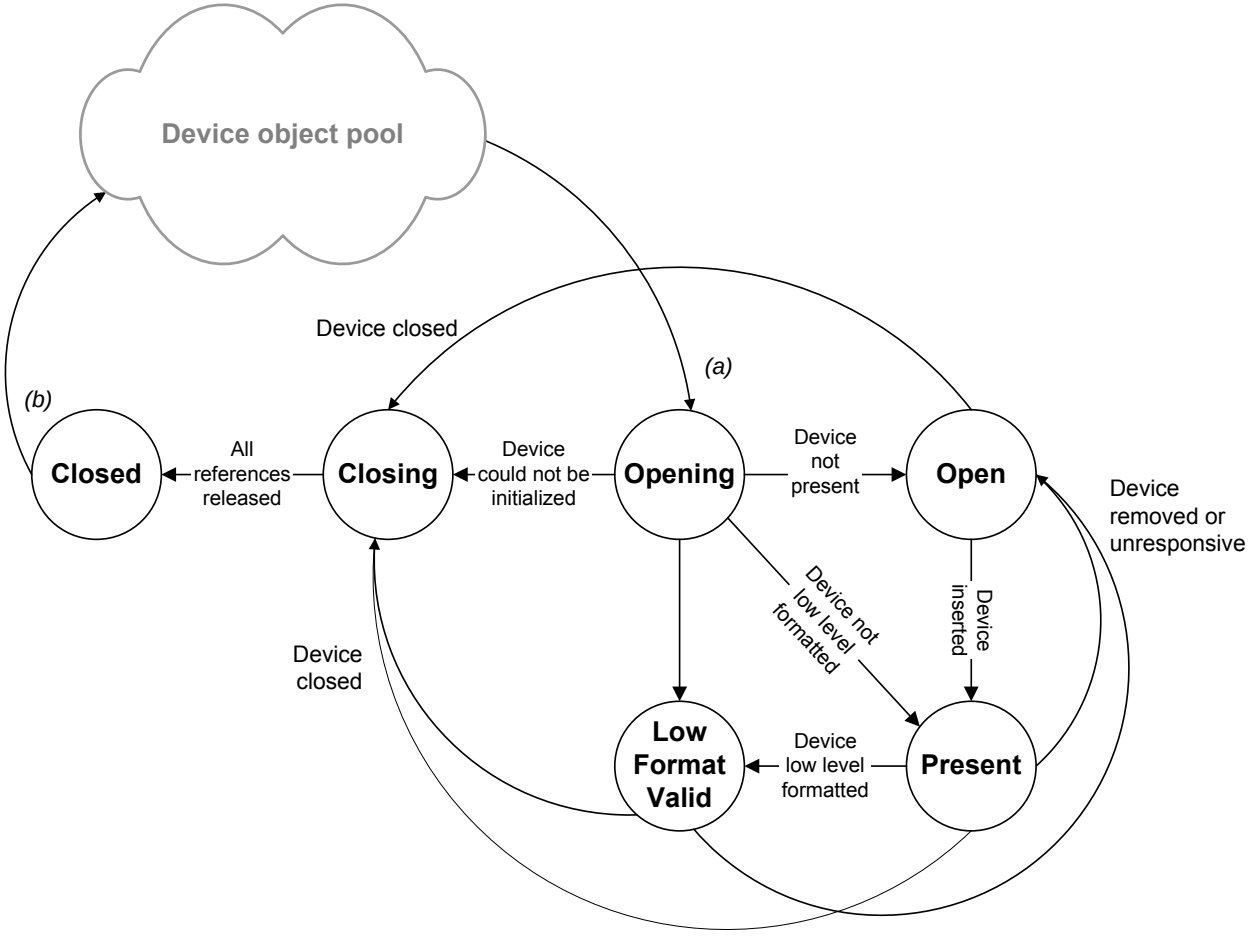


Figure 3-2. Device state transition.
Prior to `FSDev_Open()` being called (a), software is ignorant of the presence, state or characteristics of the particular device. After all references to the device are released (b), this ignorance again prevails,and any buffers or structures are freed for later use.

The return error code from this functions provides important information about the device state:

- If the return error code is `FS_ERR_NONE`, then the device is present, responsive and low-level formatted; basically, it is ready to use.

- If the return error code is `FS_ERR_DEV_INVALID_LOW_FMT`, then the device is present and responsive, but must be low-level formatted. The application should next call `FSDev_NOR_LowFmt()` for the NOR flash

- If the return error code is `FS_ERR_DEV_NOT_PRESENT`, `FS_ERR_DEV_IO` or `FS_ERR_DEV_TIMEOUT`, the device is either not present or did not respond. This is an important consideration for removable devices. It is still registered with the file system suite, and the file system will attempt to re-open the device each time the application accesses it (see Section 3.01.02 for more information).

- If any other error code is returned, the device is NOT registered with the file system. The developer should examine the error code to determine the source of the error.

3.03 Using Removable Devices

µC/FS expects that any call to a function that accesses a removable device may fail, since the device may be removed, powered off or suddenly unresponsive. If µC/FS detects such an event, the device will need to be refreshed or closed and re-opened. `FSDev_Refresh()` refreshes a device:

```
       (a)
chngd = FSDev_Refresh((CPU_CHAR *)“ide:0:”,  <-- (b) device name
                      (FS_ERR   *)&err);     <-- (c) return error
```

There are several cases to consider:

- If the return error is `FS_ERR_NONE` and the return value (a) is `DEF_YES`, then a new device (e.g., SD card) has been inserted. All files and directories that are open on volumes on the device must be closed and all volumes that are open on the device must be closed or refreshed.

- If the return error is `FS_ERR_NONE` and the return value (a) is `DEF_NO`, then the same device (e.g., SD card) is still inserted. The application can continue to access open files, directories and volumes.

- If the return error is neither `FS_ERR_NONE` nor `FS_ERR_DEV_INVALID_LOW_FMT`, then no functioning device is present. The device must be refreshed at a later time.

A device can be refreshed explicitly with `FSDev_Refresh()`; however, refresh also happens automatically. If a volume access (e.g., `FSVol_Fmt()`, `FSVol_Rd()`) , entry access (`FSEntry_Create()`, `fs_remove()`), file open (`fs_fopen()` or `FSFile_Open()`) or directory open (`fs_opendir()` or `FSDir_Open()`) is initiated on a device that was not present at the last attempted access, µC/FS attempts to refresh the device information; if that succeeds, it attempts to refresh the volume information.

Files and directories have additional behavior. If a file is opened on a volume, and the underlying device is subsequently removed or changed, all further accesses using the file API (e.g., `FSFile_Rd()`) will fail with the error code `FS_ERR_DEV_CHNGD`; all POSIX API functions will return error values The file should then be closed (to free the file structure).

Similarly, if a directory is opened on a volume, and the underlying device is subsequently removed or changed, all further `FSDir_Rd()` attempts will fail with the error code `FS_ERR_DEV_CHNGD`; `fs_readdir_r()` will return 1. The directory should then be closed (to free the directory structure).

3.04 Partitions

A device can be partitioned into two or more regions, and a file system created on one or more of these, each of which could be mounted as a volume. µC/FS can handle and make DOS-style partitions, which is a common partitioning system.

The first sector on a device with DOS-style partitions is the **Master Boot Record (MBR)**, with a partition table with four entries, each describing a partition. An MBR entry contains the start address of a partition, the number of sectors it contains and its type. The structure of a MBR entry and the MBR sector is shown in Figure 3-3.

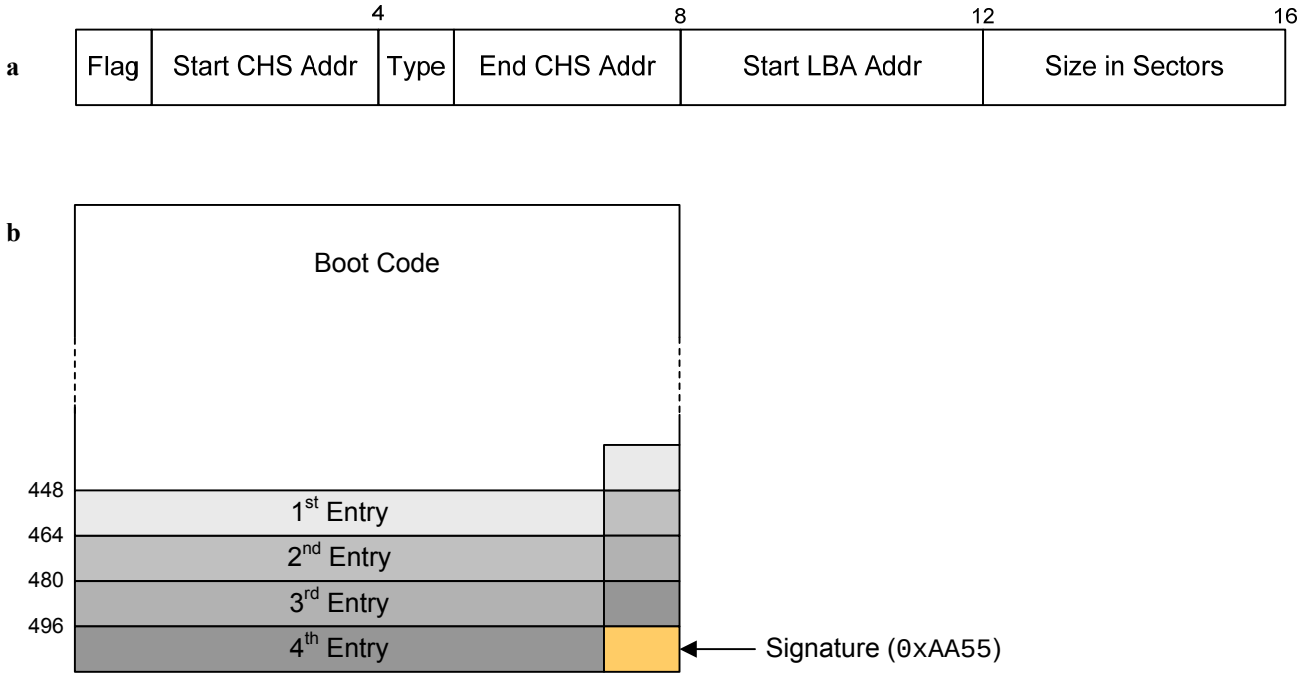


Figure 3-3. Master Boot Record.
a. Partition entry format.
b. MBR sector layout.

An application can write an MBR to a device and create an initial partition with `FSDev_PartitionInit()`. For example, if you wanted to create an initial 256-MB partition on a 1-GB device "ide:0:":

```
FSDev_PartitionInit((CPU_CHAR   *)"ide:0:",      <-- (a) device name
                    (FS_SEC_QTY  )(512 * 1024),  <-- (b) size of partition
                    (FS_ERR     *)&err);         <-- (c) return error
```

The parameters are the device name (a) and the size of the partition, in sectors (b). If (b) is 0, then the partition will take up the entire device. After this call, the device will be divided as shown in Figure 3-4. This new partition is called a **primary partition** because its entry is in the MBR. The four circles in the MBR represent the four partition entries; the one that is now used 'points to' Primary Partition 1.

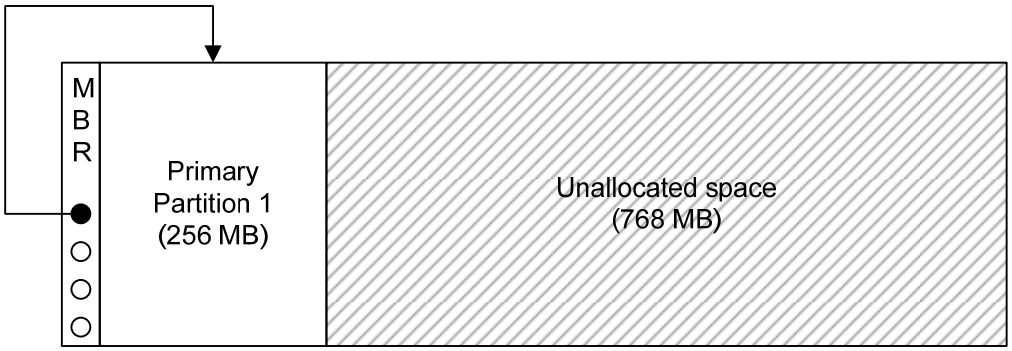


Figure 3-4. Device after partition initialization.

More partitions can now be created on the device. Since the MBR has four partition entries, three more can be made without using extended partitions (as discussed below). The function `FSDev_PartitionAdd()` should be called three times:

```
FSDev_PartitionAdd((CPU_CHAR   *)“ide:0:”,       <-- (a) device name
                   (FS_SEC_QTY  )(512 * 1024),   <-- (b) size of partition
                   (FS_ERR     *)&err);          <-- (c) return error
```

Again, the parameters are the device name (a) and the size of the partition, in sectors (b). After this has been done, the device is divided as shown in Figure 3-5.

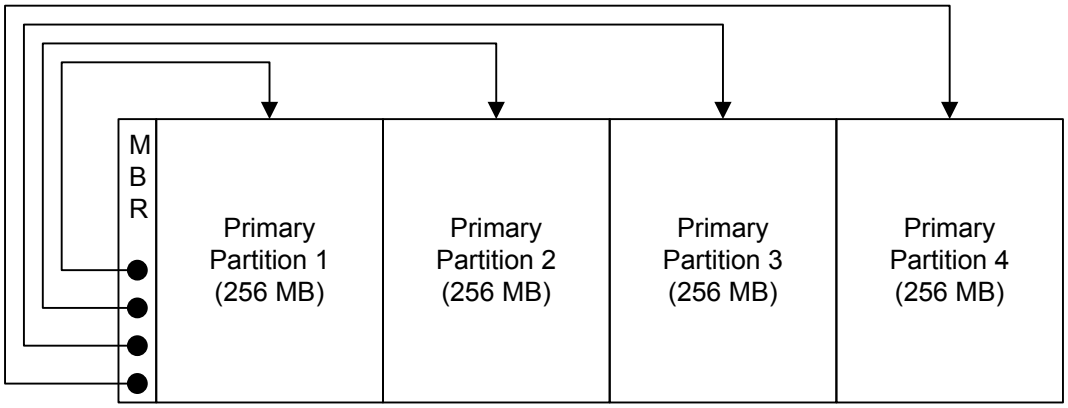


Figure 3-5. Device after four partitions have been created.

When first instituted, DOS partitioning was a simple scheme allowing up to four partitions, each with an entry in the MBR. It was later extended for larger devices requiring more with **extended partitions**, partitions that contains other partitions. The **primary extended partition** is the extended partition with its entry in the MBR; it should be the last occupied entry.

An extended partition begins with a partition table that has up to two entries (typically). The first defines a **secondary partition** which may contain a file system. The second may define another extended partition; in this case, a **secondary extended partition**, which can contain yet another secondary partition and secondary extended partition. Basically, the primary extended partition heads a linked list of partitions.

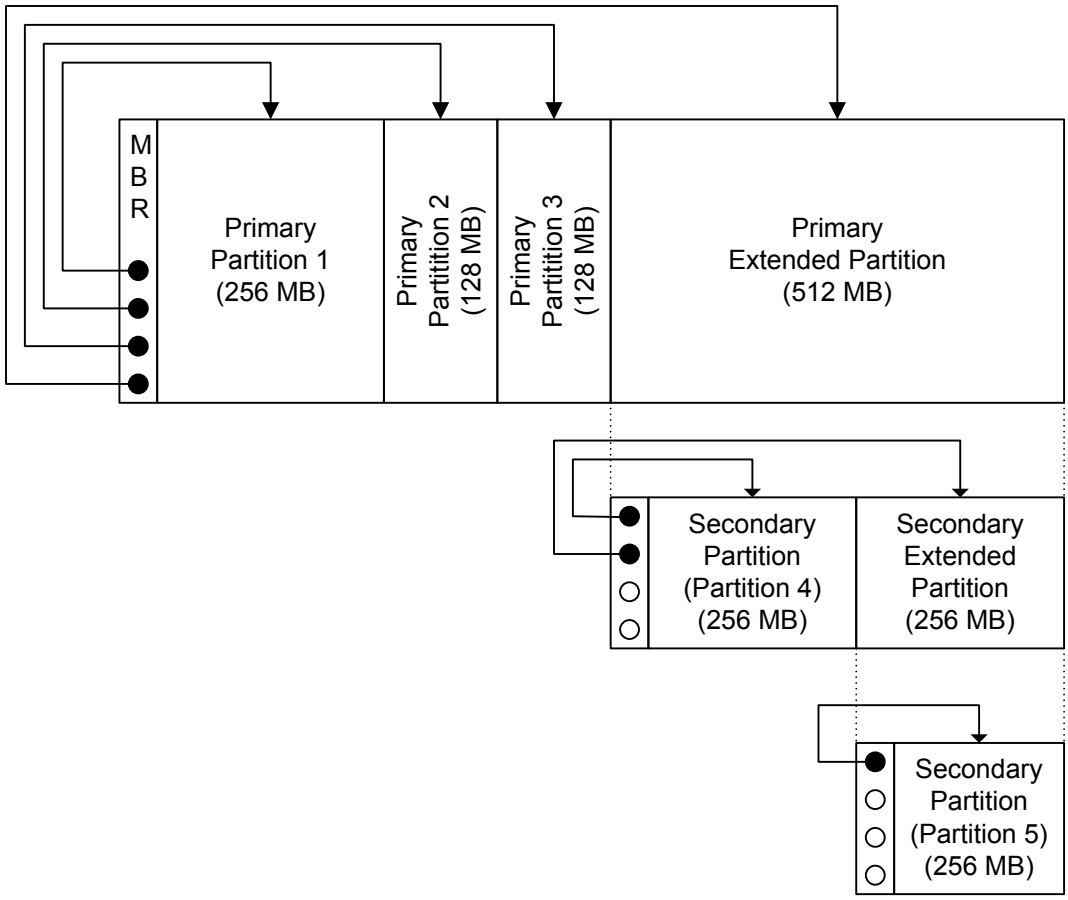


Figure 3-6. Device with five partitions.

For the moment, extended partitions are not supported in µC/FS.

3.05 Volume Operations

Five general operations can be performed on a volume:

- A volume can be **opened (mounted)**. During the opening of a volume, file system control structures are read from the underlying device, parsed and verified.

- **Files can be accessed** on a volume. A file is a linear data sequence ('file contents') associated with some logical, typically human-readable identifier ('file name'). Additional properties, such as size, update date/time and access mode (e.g., read-only, write-only, read-write) may be associated with a file. File accesses constitute reading data from files, writing data to files, creating new files, renaming files, copying files, etc. File access is accomplished via file module-level functions, which are covered in Chapter 5.

- **Directories can be accessed** on a volume. A directory is a container for files and other directories. Operations include iterating through the contents of the directory, creating new directories, renaming directories, etc. Directory access is accomplished via directory module-level functions, which are covered in Chapter 6.

- A volume can be **formatted**. (More specifically, high-level formatted.) Formatting writes the control information for a file system to the partition on which a volume is located.

- A volume can be **closed (unmounted)**. During the closing of a volume, any cached data is written to the underlying device and associated structures are freed.

For information about using volume names, see Section 2.04. For FAT-specific volume functions, see Chapter 7.

Function	Description	Valid for Unmounted Volume?
`FSVol_CacheAssign()`	Assign cache to volume.	Yes
`FSVol_CacheInvalidate()`	Invalidate cache for volume.	No
`FSVol_CacheFlush()`	Flush cache for volume.	No
`FSVol_Close()`	Close (unmount) volume.	Yes
`FSVol_Fmt()`	Format volume.	Yes
`FSVol_IsMounted()`	Determine whether volume is mounted.	Yes
`FSVol_LabelGet()`	Get volume label.	No
`FSVol_LabelSet()`	Set volume label.	No
`FSVol_Open()`	Open (mount) volume.	-----
`FSVol_Query()`	Get volume information.	Yes
`FSVol_Rd()`	Read sector on volume.	No
`FSVol_Refresh()`	Refresh a volume.	No
`FSVol_Wr()`	Write sector on volume.	No

Table 3-2. Volume API Functions

3.06 Using Volumes

A volume is opened with `FSVol_Open()`:

```
FSVol_Open((CPU_CHAR         *)“ide:0:”,   <-- (a) volume name
           (CPU_CHAR         *)“ide:0:”,   <-- (b) device name
           (FS_PARTITION_NBR *) 0,         <-- (c) partition number
           (FS_ERR           *)&err);      <-- (d) return error
```

The parameters are the volume name (a), the device name (b) and the partition that will be opened (c). There is no restriction on the volume name (a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partitioned, then the partition number (c) should be zero.

The return error code from this function provides important information about the volume state:

- If the return error code is `FS_ERR_NONE`, then the volume has been mounted and is ready to use.
- If the return error code is `FS_ERR_PARTITION_NOT_FOUND`, then no valid file system could be found on the device, or the specified partition does not exist. The device may need to be formatted (see below).
- If the return error code is `FS_ERR_DEV`, `FS_ERR_DEV_NOT_PRESENT`, `FS_ERR_DEV_IO` or `FS_ERR_DEV_TIMEOUT`, the device is either not present or did not respond. This is an important consideration for removable devices. The volume is still registered with the file system suite, and the file system will attempt to re-open the volume each time the application accesses it (see Section 3.01.02 for more information).
- If any other error code is returned, the volume is NOT registered with the file system. The developer should examine the error code to determine the source of the error.

`FSVol_Fmt()` formats a device, (re-)initializing the file system on the device:

```
FSVol_Fmt((CPU_CHAR *)“ide:0:”,  <-- (a) volume name
          (void     *) 0,        <-- (b) pointer to system configuration
          (FS_ERR   *)&err);     <-- (c) return error
```

The parameters are the volume name (a) and a pointer to file system-specific configuration (b). The configuration is not required; if you are willing to accept the default format, a NULL pointer should be passed. Alternatively, the exact properties of the file system can be configured by passing a pointer to a `FS_FAT_FMT_CFG` structure as the second argument. For more information about the `FS_FAT_FMT_CFG` structure, see Section C.02.

3.07 Using Volume Cache

File accesses often incur repeated reading of the same volume sectors. On a FAT volume, these may be sectors in the root directory, the area of the file allocation table (FAT) from which clusters are being allocated or data from important (often-read) files. A cache wedged between the system driver and volume layers (as shown in Figure 3-7) will eliminate many unnecessary device accesses. Sector data is stored upon first read or write. Further reads return the cached data; further writes update the cache entry and, possibly, the data on the volume (depending on the cache mode).

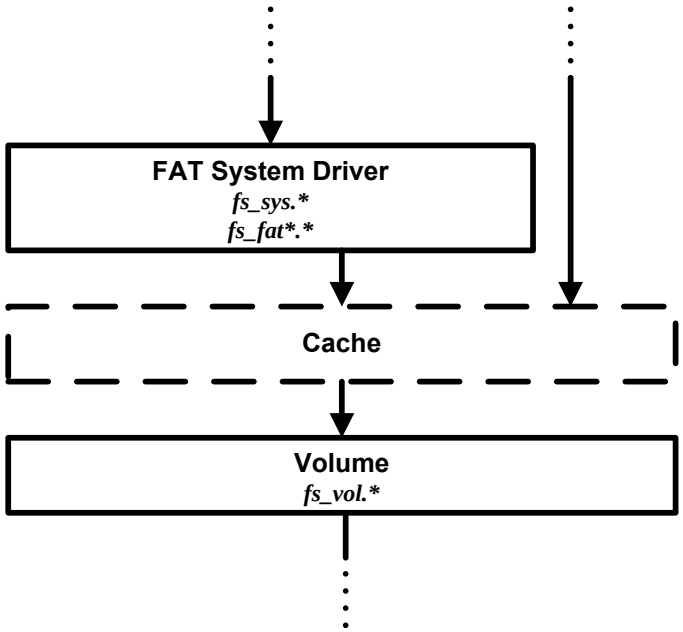


Figure 3-7. Volume cache architecture.

A cache is defined by three parameters: size, sector type allocation and mode. The **size** of the cache is the number of sectors that will fit into it at any time. Every sector is classified according to its type, either management, directory or file; the **sector type allocation** determines the percentage of the cache that will be devoted to each type. The **mode** determines when cache entries are created (i.e., when sectors are cached) and what happens upon write, as summarized in Table 3-3.

Cache Mode	Description	Cache Mode `#define`
Read cache	Sectors cached upon read; never cached upon write.	`FS_VOL_CACHE_MODE_RD`
Write-through cache	Sectors cached upon read and write; data on volume always updated upon write.	`FS_VOL_CACHE_MODE_WR_THROUGH`
Write-back cache	Sectors cached upon read and write; data on volume never updated upon write.	`FS_VOL_CACHE_MODE_WR_BACK`

Table 3-3. Cache types

File access presents a special case. When a file is opened with a combination of `FS_FILE_ACCESS_MODE_WR` and `FS_FILE_ACCESS_MODE_CACHED` the update of the directory sector will be delayed until the file is closed.

```
        pfile = FSFile_Open("\file.txt",
                            FS_FILE_ACCESS_MODE_WR |
                            FS_FILE_ACCESS_MODE_CACHED,
                           &err);
```

For files in read or write mode, data from the file will be cached. For files in write mode, the update of the directory sector will be delayed until the file is closed.

3.07.01 Choosing Cache Parameters

The following is an example using the cache for the volume "sdcard:0:". The cache is used in write back mode, and the cache parameters are:
25 % of cache size is used for management sector, 15 % is used for directories sectors and the remaining (60 %) is used for file sectors.

```
FSVol_CacheAssign ((CPU_CHAR         *)"sdcard:0:",               <-- volume name
                   (FS_VOL_CACHE_API *) NULL,                     <-- pointer to volume cache API
                   (void             *)&CACHE_BUF[0],             <-- pointer to the cache buffer
                   (CPU_INT32U        ) CACHE_BUF_LEN,            <-- cache buffer size in bytes
                   (CPU_INT08U        ) 25,                       <-- see (1)
                   (CPU_INT08U        ) 15,                       <-- see (2)
                   (FS_FLAGS          ) FS_VOL_CACHE_MODE_WR_BACK,<-- cache mode
                   (FS_ERR           *)&err);                     <-- used for error code

if (err != FS_ERR_NONE) {
    APP_TRACE_INFO ((" Error : could not assign Volume cache"));
    return;
}

pfile = FSFile_Open("sdcard:0:\\file.txt",
                    FS_FILE_ACCESS_MODE_WR |
                    FS_FILE_ACCESS_MODE_CACHED,
                   &err);

if (pFile == (FS_FILE *)0) {
    return;
}

/*
   DO THE WRITE OPERATIONS TO THE FILE
*/

FSFile_Close (pFile, &err);

FSVol_CacheFlush ("sdcard:0:", &err);                             <-- Flush volume cache.
```

(1) Percent of cache buffer dedicated to management sectors.
(2) Percent of cache buffer dedicated to directory sectors.

The application using µC/FS volume cache should vary the third and fourth parameters passed to **`FSVol_CacheAssign(),`** and select the values that give the best performance.

For an efficient cache usage, it is better to do not allocate space in the cache for sectors of type file when the write size is greater than sector size.

When the cache is used in write back mode, all cache dirty sectors will be updated on the media storage only when the cache is flushed..

3.07.02 Other Caching & Buffering Mechanisms

Volume cache is just one of several important caching mechanisms, which should be balanced for optimal performance within the bounds of platform resources. The second important software mechanism is the file buffer (see Section 4.02.03), which makes file accesses more efficient by buffering data so a full sector's worth will be read or written.

Individual devices or drivers may also integrate a cache. Standard hard drives overcome long seek times by buffering extra data upon read (in anticipation of future requests) or clumping writes to eliminate unnecessary movement. The latter action can be particularly powerful, but since it may involve re-ordering the sequence of sector writes will eliminate any guarantee of fail-safety of most file systems. For that reason, write cache in most storage devices should be disabled.

A driver may implement a buffer to reduce apparent write latency. Before a write can occur to a flash medium, the driver must find a free (erased) area of a block; occasionally, a block will need to be erased to make room for the next write. Incoming data can be buffered while the long erase occurs in the background, thereby uncoupling the application's wait time from the real maximum flash write time.

The ideal system might use both volume cache and file buffers. A volume cache is most powerful when confined to the sector types most subject to repeated reads: management and directory. Caching of files, if enabled, should be limited to important (often-read) files. File buffers are more flexible, since they cater to the many applications that find small reads and writes more convenient than those of full sectors.

Chapter

4

POSIX API

The best-known API for accessing and managing files and directories is specified within the POSIX standard (IEEE Std 1003.1). The basis of some of this functionality, in particular buffered input/output, lies in the ISO C standard (ISO/IEC 9899), though many extensions provide new features and clarify existing behaviors. Functions and macros prototyped in four header files are of particular importance:

- ***stdio.h***. Standard buffered input/output (`fopen()`, `fread()`, etc), operating on `FILE` objects.
- ***dirent.h***. Directory accesses (`opendir()`, `readdir()`, etc), operating on `DIR` objects.
- ***unistd.h***. Miscellaneous functions, including working directory management (`chdir()`, `getcwd()`), `ftruncate()` and `rmdir()`.
- ***sys/stat.h***. File statistics functions and `mkdir()`.

µC/FS provides a POSIX-compatible API based on a subset of the functions in these four header files. To avoid conflicts with the user compilation environment, files, functions and objects are renamed:

- All functions begin with '`fs_`'. For example, `fopen()` is renamed `fs_fopen()`, `opendir()` is renamed `fs_opendir()`, `getcwd()` is renamed `fs_getcwd()`, etc.
- All objects begin with '`FS_`'. So `fs_fopen()` returns a pointer to a `FS_FILE` and `fs_opendir()` returns a pointer to a `FS_DIR`.
- Some argument types are renamed. For example, the second and third parameters of `fs_fread()` are typed `fs_size_t` to avoid conflicting with other `size_t` definitions.

4.01 Supported Functions

The supported POSIX functions are listed in the table below. These are divided into four groups. First, the functions which operate on file objects (`FS_FILE`s) are grouped under **file access (or simply file) functions**. An application stores information in a file system by creating a file or appending new information to an existing file. At a later time, this information may be retrieved by reading the file. Other functions support these capabilities; for example, the application can move to a specified location in the file or query the file system to get information about the file.

A separate set of **file operations (or entry) functions** manage the files and directories available on the system. Using these functions, the application can create, delete and rename files and directories.

The entries within a directory can be traversed using the **directory access (or simply directory) functions**, which operate on directory objects (`FS_DIR`s). The name and properties of the entries are returned within a `struct fs_dirent` structure.

The final group of functions is the **working directory functions**.

For information about using file and path names, see Section 2.05.

Function	POSIX Equivalent
`fs_asctime_r()`	`asctime_r()`
`fs_chdir()`	`chdir()`
`fs_clearerr()`	`clearerr()`
`fs_closedir()`	`closedir()`
`fs_ctime_r()`	`ctime_r()`
`fs_fclose()`	`fclose()`
`fs_feof()`	`feof()`
`fs_ferror()`	`ferror()`
`fs_fflush()`	`fflush()`
`fs_fgetpos()`	`fgetpos()`
`fs_flockfile()`	`flockfile()`
`fs_fopen()`	`fopen()`
`fs_fread()`	`fread()`
`fs_fseek()`	`fseek()`
`fs_fsetpos()`	`fsetpos()`
`fs_fstat()`	`fstat()`
`fs_ftell()`	`ftell()`

Function	POSIX Equivalent
`fs_ftruncate()`	`ftruncate()`
`fs_ftrylockfile()`	`ftrylockfile()`
`fs_funlockfile()`	`funlockfile()`
`fs_fwrite()`	`fwrite()`
`fs_getcwd()`	`getcwd()`
`fs_localtime_r()`	`localtime_r()`
`fs_mkdir()`	`mkdir()`
`fs_mktime()`	`mktime()`
`fs_rewind()`	`rewind()`
`fs_opendir()`	`opendir()`
`fs_readdir_r()`	`readdir_r()`
`fs_remove()`	`remove()`
`fs_rename()`	`rename()`
`fs_rmdir()`	`rmdir()`
`fs_setbuf()`	`setbuf()`
`fs_setvbuf()`	`setvbuf()`
`fs_stat()`	`stat()`

Table 4-1. POSIX API functions.

4.02 Working Directory Functions

Normally, all file or directory paths must be absolute, either on the default volume or on an explicitly-specified volume:

```
p_file1 = fs_fopen("\\file.txt", "r");            /* File on default volume */

p_file2 = fs_fopen("sdcard:0:\\file.txt", "r");   /* File on explicitly-specified volume */
```

If working directory functionality is enabled, paths may be specified relative to the working directory of the current task:

```
p_file2 = fs_fopen("file.txt", "r");              /* File in working directory */

p_file1 = fs_fopen("..\\file.txt", "r");          /* File in parent of working directory */
```

The two standard special path components are supported. The path component ".." moves to the parent of the current working directory. The path component "." makes no change; essentially, it means the current working directory.

`fs_chdir()` is used to set the working directory. If a relative path is employed before any working directory is set, the working directory is taken as the root directory of the default volume.

The application can get the working directory with `fs_getcwd()`. A terminal interface may use this function to implement an equivalent to the standard `pwd` (print working directory) command, while calling `fs_chdir()` to carry out a `cd` operation. If working directories are enabled, the µC/Shell commands for µC/FS manipulate and access the working directory with `fs_chdir()` and `fs_getcwd()` (see also Appendix E).

4.03 File Access Functions

The file access functions provide an API for performing a sequence of operations on a file located on a volume's file system. The file object pointer returned when a file is opened is passed as an argument of all file access function, and the file object so referenced maintains information about the actual file (on the volume) and the state of the file access. The file access state includes the file position (the next place data will be read/written), error conditions and (if file buffering is enabled) the state of any file buffer.

As data is read from or written to a file, the file position is incremented by the number of bytes transferred from/to the volume. The file position may also be directly manipulated by the application using the position set function (`fs_fsetpos()`), and the current absolute file position may be gotten with the position get function (`fs_fgetpos()`), to be later used with the position set function.

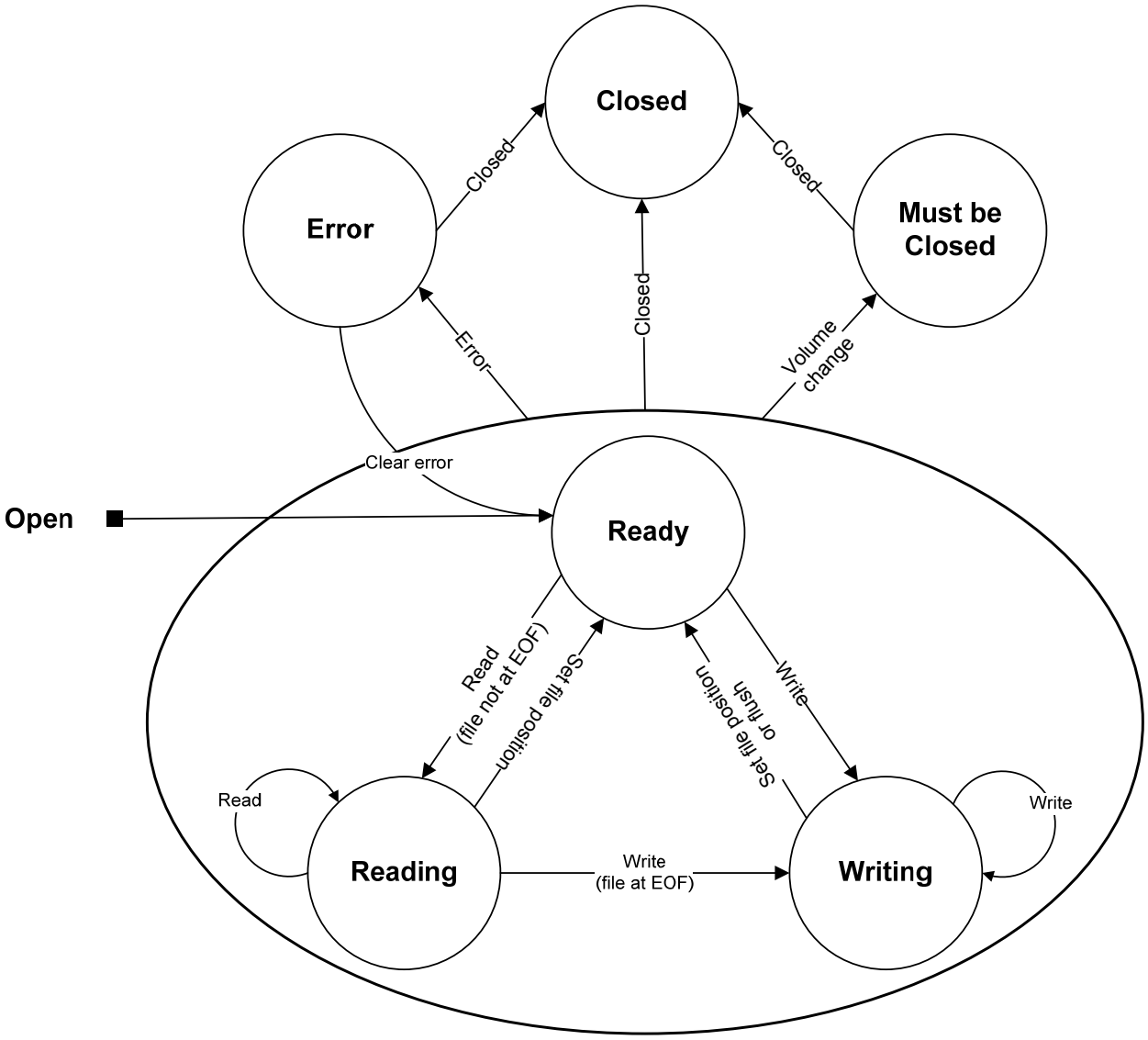


Figure 4-1. File state transitions.

The file maintains flags that reflect errors encountered in the previous file access, and subsequent accesses will fail (under certain conditions outlined here) unless these flags are explicitly cleared (using `fs_clearerr()`). There are actually two sets of flags. One reflects whether the file encountered the end-of-file (EOF) during the previous access, and if this is set, writes will not fail, but reads will fail. The other reflects device errors, and no subsequent file access will succeed (except file close) unless this is first cleared. The functions `fs_ferror()` and `fs_feof()` can be used to get the state of device error and EOF conditions, respectively.

If file buffering is enabled (`FS_CFG_FILE_BUF_EN` is `DEF_ENABLED`), then input/output buffering capabilities can be used to increase the efficiency of file reads and writes. A buffer can be assigned to a file using `fs_setbuf()`or `fs_setvbuf()`; the contents of the buffer can be flushed to the storage device using `fs_fflush()`.

If a file is shared between several tasks in an application, a file lock can be employed to guarantee that a series of file operations are executed atomically. `fs_flockfile()` (or its non-blocking equivalent `fs_ftrylockfile()`) acquires the lock for a task (if it does not already own it). Accesses from other tasks will be blocked until a `fs_funlockfile()` is called. This functionality is available if `FS_CFG_FILE_LOCK_EN` is `DEF_ENABLED`.

4.03.01 Opening, Reading & Writing Files

When an application needs to access a file, it must first open it using `fs_fopen()`:

```
file pointer -->  p_file = fs_fopen("\\file.txt", <-- file name
                                    "w+");       <-- mode string

                  if (p_file == (FS_FILE *)0) {
                      /* $$$$ Handle error */
                  }
```

The return value of this function should always be verified as non-NULL before the application proceeds to access the file. The first argument of this function is the path of the file; if working directories are disabled, this must be the absolute file path, beginning with either a volume name or a '\' (see Section 2.05). The second argument of this function is a string indicating the mode of the file; this must be one of the strings shown in the table below. Note that in all instances, the 'b' (binary) option has no affect on the behavior of file accesses.

`fs_fopen()` Mode String	Read?	Write?	Truncate?	Create?	Append?
"r" or "rb"	Yes	No	No	No	No
"w" or "wb"	No	Yes	Yes	Yes	No
"a" or "ab"	No	Yes	No	Yes	Yes
"r+" or "rb+" or "r+b"	Yes	Yes	No	No	No
"w+" or "wb+" or "w+b"	Yes	Yes	Yes	Yes	No
"a+" or "ab+" or "a+b"	Yes	Yes	No	Yes	Yes

Table 4-2. `fs_fopen()` mode strings interpretations.

After a file is opened, any of the file access functions valid for that its mode can be called. The most commonly used functions are `fs_fread()` and `fs_fwrite()`, which read or write a certain number of 'items' from a file:

```
number of items read -->  cnt = fs_fread(p_buf,   <-- pointer to buffer
                                         1,       <-- size of each item
                                         100,     <-- number of items
                                         p_file); <-- pointer to file
```

The return value, the number of items read (or written), should be less than or equal to the third argument. If the operation is a read, this value may be less than the third argument for one of two reasons. First, the file could have encountered the end-of-file (EOF), which means that there is no more data in the file. Second, the device could have been removed, or some other error could have prevented the operation. To diagnose the cause, the `fs_feof()` function should be used. This function returns a non-zero value if the file has encountered the EOF.

Once the file access is complete, the file MUST be closed; if an application fails to close files, then the file system suite resources such as file objects may be depleted.

An example of reading a file is given in Listing 4-1.

Listing 4-1. Example file read.

```
void  App_Fnct (void)
{
    FS_FILE         *p_file;
    fs_size_t        cnt;
    unsigned  char   buf[50];

    .
    .
    .


    p_file = fs_fopen("\\file.txt", "r");          /* Open file.                              */

    if (p_file != (FS_FILE *)0) {                  /* If file is opened ...                   */
                                                   /* ... read from file.                     */
        do {
            cnt = fs_fread(&buf[0], 1, sizeof(buf), p_file);
            if (cnt > 0) {
                APP_TRACE_INFO(("Read %d bytes.\r\n", cnt));
            }
        } while (cnt >= sizeof(buf));

        eof = fs_feof(p_file);                     /* Chk for EOF.                            */
        if (eof != 0) {                            /* See Note #1.                            */
            APP_TRACE_INFO(("Reached EOF.\r\n"));

        } else {
            err = fs_ferror(p_file);               /* Chk for error.                          */
            if (err != 0) {                        /* See Note #2.                            */
                APP_TRACE_INFO(("Read error.\r\n"));
            }
        }

        fs_fclose(p_file);                         /* Close file.                             */
    } else {
        APP_TRACE_INFO(("Could not open \"\\file.txt\".\r\n"));
    }

    .
    .
    .
}
```

L4-1(1) To determine whether a file read terminates because of reaching the EOF or a device error/removal, the EOF condition should be checked using `fs_feof()`.

L4-1(2) In most situations, either the EOF or the error indicator will be set on the file if the return value of `fs_fread()` is smaller than the buffer size. Consequently, this check is unnecessary.

4.03.02 Getting or Setting the File Position

Another common operation is getting or setting the file position. The `fs_fgetpos()` and `fs_fsetpos()` allow the application to 'store' a file location, continue reading or writing the file, and then go back to that place at a later time. An example of using file position get and set is given in Listing 4-2.

Listing 4-2. Example file position set/get.

```
void  App_Fnct (void)
{
    FS_FILE    *p_file;
    fs_fpos_t   pos;
    int         err;

    .
    .
    .

    p_file = fs_fopen("\file.txt", "r");           /* Open file ...                          */
    if (p_file == (FS_FILE *)0) {
        APP_TRACE_INFO(("Could not open file."));
        return;
    }

    .
    .                                              /* ... read from file.                    */
    .

    err    = fs_fgetpos(p_file, &pos);             /* Save file position ...                 */
    if (err != 0) {
        APP_TRACE_INFO(("Could not get file position."));
        return;
    }

    .
    .                                              /* ... read some more from file.          */
    .

    err    = fs_fsetpos(p_file, &pos);             /* Set file to saved position ...         */
    if (err != 0) {
        APP_TRACE_INFO(("Could not set file position."));
        return;
    }

    .
    .                                              /* ... read some more from file.          */
    .

    FS_fclose(p_file);                             /* When finished, close file.             */

    .
    .
    .
}
```

4.03.03 Configuring a File Buffer

In order to increase the efficiency of file reads and writes, input/output buffering capabilities are provided. Without an assigned buffer, reads and writes will be immediately performed within `fs_fread()` and `fs_fwrite()`. Once a buffer has been assigned, data will always be read from or written to the buffer; device access will only occur once the file position moves beyond the window represented by the buffer.

`fs_setbuf()` and `fs_setvbuf()` assign the buffer to a file. The contents of the buffer can be flushed to the storage device with `fs_fflush()`. If a buffer is assigned to a file that was opened in update (read/write) mode, then a write may only be followed by a read if the buffer has been flushed (by calling `fs_fflush()` or a file positioning function). A read may be followed by a write only if the buffer has been flushed, except when the read encountered the end-of-file, in which case a write may happen immediately. The buffer is automatically flushed when the file is closed.

File buffering is particularly important when data is written in small chunks to a medium with slow write time or limited endurance. An example is NOR flash, or even NAND flash, where write times are much slower than read times, and the lifetime of device is constrained by limits on the number of times each block can be erased and programmed.

Listing 4-3. Example file buffer usage.

```
static  CPU_INT32U  App_FileBuf[512 / 4];          /* Define file buffer.                    */

void  App_Fnct (void)
{
    CPU_INT08U  data1[50];
    .
    .
    .

    p_file = FS_fopen("\\file.txt", "w");
    if (p_file != (FS_FILE *)0) {
                                                   /* Set buffer (see Note #1).              */
        fs_setvbuf(p_file, (void *)App_FileBuf, FS__IOFBF, sizeof(App_FileBuf));

        .
        .
        .

        fs_fflush(p_file);                         /* Make sure data is written to file.     */

        .
        .
        .

        fs_fclose(p_file);                         /* When finished, close file.             */
    }

    .
    .
    .
}
```

L4-3(1) The buffer MUST be assigned immediately after opening the file. An attempt to set the buffer after read or writing the file will fail.

L4-3(2) While it is not necessary to flush the buffer before closing the file, some applications may want to make sure at certain points that all previously written data is stored on the device before writing more.

4.03.04 Diagnosing a File Error

The file maintains flags that reflect errors encountered in the previous file access, and subsequent accesses will fail (under certain conditions outlined here) unless these flags are explicitly cleared (using `fs_clearerr()`). There are actually two sets of flags. One reflects whether the file encountered the end-of-file (EOF) during the previous access, and if this is set, writes will not fail, but reads will fail. The other reflects device errors, and no subsequent file access will succeed (except file close) unless this is first cleared. The functions `fs_ferror()` and `fs_feof()` can be used to get the state of device error and EOF conditions, respectively.

4.03.05 Atomic File Operations Using File Lock

If a file is shared between several tasks in an application, the file lock can be employed to guarantee that a series of file operations are executed atomically. `fs_flockfile()` (or its non-blocking equivalent `fs_ftrylockfile()`) acquires the lock for a task (if it does not already own it). Accesses from other tasks will be blocked until `fs_funlockfile()` is called.

Each file actually has a lock count associated with it. This allows nested calls by a task to acquire a file lock; each of those calls must be matched with a call to `fs_funlockfile()`.

Listing 4-4. Example file lock usage.

```
void  App_Fnct (void)
{
    unsigned  char  data1[50];
    unsigned  char  data2[10];
    .
    .
    .

    if (App_FilePtr != (FS_FILE *)0) {
        fs_flockfile(App_FilePtr);                  /* Lock file.                          */
                                                    /* See Note #1.                        */

                                                    /* Wr data atomically.                 */
        fs_fwrite(data1, 1, sizeof(data1), App_FilePtr);
        fs_fwrite(data2, 1, sizeof(data1), App_FilePtr);

        fs_funlockfile(App_FilePtr);                /* Unlock file.                        */
    }

    .
    .
    .
}
```

L4-4(1) `fs_flockfile()` will block the calling task until the file is available. If the task must write to the file only if no other task is currently accessing it, the non-blocking function `fs_funlockfile()` can be used.

4.04 Directory Access Functions

The directory access functions provide an API for iterating through the entries within a directory. The `fs_opendir()` function initiates this procedure, and each subsequent call to `fs_readdir_r()` (until all entries have been examined) returns information about a particular entry in a `struct fs_dirent`. The `fs_closedir()` function releases any file system structures and locks.

Listing 4-5 gives an example using the directory access functions to list the files in a directory. An example result of listing a directory is shown in Figure 4-1.

Listing 4-5. Example directory listing.

```
void  App_Fnct (void)
{
    FS_DIR             *p_dir;
    struct  fs_dirent   dirent;
    struct  fs_dirent  *p_dirent;
    char                str[50];
    char               *p_cwd_path;
    fs_time_t           ts;
    .
    .
    .

    p_dir = fs_opendir(p_cwd_path);                              /* Open dir.                  */

    if (p_dir != (FS_DIR *)0) {
        (void)fs_readdir_r(pdir, &dirent, &p_dirent);            /* Rd first dir entry.        */

        if (p_dirent == (FS_DIRENT *)0) {                        /* If NULL ... dir is empty.  */
            APP_TRACE_INFO((“Empty dir: %s.\r\n”, p_cwd_path));

        } else {                                                 /* Fmt info for each entry.   */
            Str_Copy(str, "-r--r—r--                      :   ");
            while (p_dirent != (struct dirent *)0) {
                                                                 /* Chk if file is dir.        */
                if (DEF_BIT_IS_SET(dirent.Info.Attrib, FS_ENTRY_ATTRIB_DIR) == DEF_YES) {
                    str[0] = ‘d’;
                }
                                                                 /* Chk if file is rd only.    */
                if (DEF_BIT_IS_SET(dirent.Info.Attrib, FS_ENTRY_ATTRIB_WR) == DEF_YES) {
                    str[2] = ‘w’;
                    str[5] = ‘w’;
                    str[8] = ‘w’;
                }
                                                                 /* Get file size.             */
                if (p_dirent->Info.Size == 0) {
                    if (DEF_BIT_IS_CLR(dirent.Info.Attrib, FS_ENTRY_ATTRIB_DIR) == DEF_YES) {
                        Str_Copy(&str[11],"          0");
                    }
                } else {
                    Str_FmtNbr_Int32U(dirent.Info.Size,
                                      10, 10, ‘0’, DEF_NO, DEF_NO, &str[11]);
                }
                                                                 /* Get file date/time.        */
                if (p_dirent->Info.DateTimeCreate.Month != 0) {
                    Str_Copy(&str[22],
                         (CPU_CHAR *)App_MonthNames[dirent.Info.DateTimeCreate.Month - 1]);
                    Str_FmtNbr_Int32U(dirent.Info.DateTimeWr.Day,
                                      2, 10, ‘ ’, DEF_NO, DEF_NO, &str[26]);
                    Str_FmtNbr_Int32U(dirent.Info.DateTimeWr.Hour,
                                      2, 10, ‘ ’, DEF_NO, DEF_NO, &str[29]);
                    Str_FmtNbr_Int32U(dirent.Info.DateTimeWr.Minute,
                                      2, 10, ‘ ’, DEF_NO, DEF_NO, &str[32]);
                }
```

```
                                                        /* Output info for entry.      */
                APP_TRACE_INFO(("%s%s\r\n", str, dirent.Name));

                (void)fs_readdir_r(pdir, &dirent, &p_dirent);  /* Rd next dir entry.          */
            }
        }

        fs_closedir(p_dir);                                    /* Close dir.                  */



                                                       /* If dir could not be opened ... */
    } else {                                           /* ... dir does not exist.        */
        APP_TRACE_INFO(("Dir does not exist: %s.\r\n", p_cwd_path));
    }
    .
    .
    .
}
```

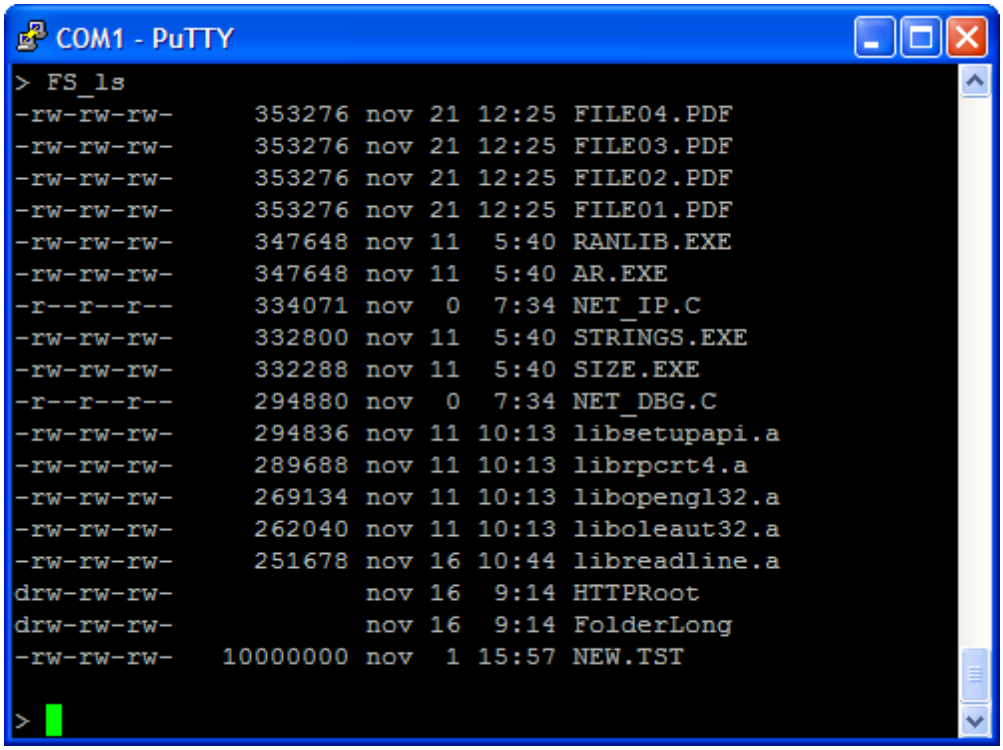


Figure 4-1. Directory Listing Output (example).

The second argument `fs_readdir_r()`, is a pointer to a `struct fs_dirent`, which has two members. The first is `Name`, which holds the name of the entry; the second is `Info`, which has file information. For more information about the struct `fs_dirent` structure, see Section C.07

4.05 Entry Access Functions

The entry access functions provide an API for performing single operations on file system entries (files and directories), such as renaming or deleting a file. Each of these operations is atomic; consequently, in the absence of device access errors, either the operation will have completed or no change to the storage device will have been made upon function return.

A new directory can be created with `fs_mkdir()` or an existing file or directory deleted or renamed (with `fs_remove()` or `fs_rename()`).

Chapter

5

Files

An application stores information in a file system by creating a file or appending new information to an existing file. At a later time, this information may be retrieved by reading the file. Other functions support these capabilities; for example, the application can move to a specified location in the file or query the file system to get information about the file. These functions, which operate on file structures (`FS_FILE`s), are grouped under **file access (or simply file) functions**. The available file functions are listed in Table 5-1.

A separate set of **file operations (or entry) functions** manage the files and directories available on the system. Using these functions, the application can copy, create, delete and rename files, and get and set a file or directory's attributes and date/time. The available entry functions are listed in Table 5-2.

The entry functions and the FSFile_`Open()` function accept **full file paths**. For information about using file and path names, see Section 2.05.

The functions listed in Table 5-1 and Table 5-2 are core functions in the file access module (`FSFile_####()` functions) and entry module (`FSEntry_####()` functions). These are matched, in most cases, by API level functions that correspond to standard C or POSIX functions. The core and API functions provide basically the same functionality; the benefits of the former are enhanced capabilities, a consistent interface and meaningful return error codes.

5.01 File Access Functions

The file access functions provide an API for performing a sequence of operations on a file located on a volume's file system. The file object pointer returned when a file is opened is passed as the first argument of all file access functions (a characteristic which distinguishes these from the entry access functions), and the file object so referenced maintains information about the actual file (on the volume) and the state of the file access. The file access state includes the file position (the next place data will be read/written), error conditions and (if file buffering is enabled) the state of any file buffer.

Function	Description
`FSFile_BufAssign()`	Assign buffer to a file.
`FSFile_BufFlush()`	Write buffered data to volume.
`FSFile_Close()`	Close a file.
`FSFile_ClrErr()`	Clear error(s) on a file.
`FSFile_IsEOF()`	Determine whether a file is at EOF.
`FSFile_IsErr()`	Determine whether error occurred on a file.
`FSFile_LockGet()`	Acquire task ownership of a file.
`FSFile_LockSet()`	Release task ownership of a file.
`FSFile_LockAccept()`	Acquire task ownership of a file (if available).
`FSFile_Open()`	Open a file.
`FSFile_PosGet()`	Get file position.
`FSFile_PosSet()`	Set file position.
`FSFile_Query()`	Get information about a file.
`FSFile_Rd()`	Read from a file.
`FSFile_Truncate()`	Truncate a file.
`FSFile_Wr()`	Write to a file.

Table 5-1. File Access Functions

5.01.01 Opening Files

When an application needs to access a file, it must first open it using `fs_fopen()` or `FSFile_Open()`. For most applications, the former with its familiar interface suffices. In some cases, the flexibility of the latter is demanded:

```
file ptr --> p_file = FSFile_Open ("\\file.txt",           <-- file name
                                   FS_FILE_ACCESS_MODE_RD, <-- access mode
                                   &err);                  <-- return error

             if (p_file == (FS_FILE *)0) {
                 /* $$$$ Handle error */
             }
```

The return value of this function should always be verified as non-NULL before the application proceeds to access the file. The second argument to this function is a logical OR of mode flags:

`FS_FILE_ACCESS_MODE_RD`	File opened for reads.
`FS_FILE_ACCESS_MODE_WR`	File opened for writes.
`FS_FILE_ACCESS_MODE_CREATE`	File will be created, if necessary.
`FS_FILE_ACCESS_MODE_TRUNC`	File length will be truncated to 0.
`FS_FILE_ACCESS_MODE_APPEND`	All writes will be performed at EOF.
`FS_FILE_ACCESS_MODE_EXCL`	File will be opened if and only if it does not already exist.
`FS_FILE_ACCESS_MODE_CACHED`	File data will be cached.

For example, if you wanted to create a file to write to if and only if it does not exist, you would use the flags

```
FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE |
                                    FS_FILE_ACCESS_MODE_EXCL
```

It is impossible to do this in a single, atomic operation using `fs_fopen()`.

The table below lists the mode flag equivalents of the `fs_fopen()` mode strings.

"`r`" or "`rb`"	`FS_FILE_ACCESS_MODE_RD`

"`w`" or "`wb`"	`FS_FILE_ACCESS_MODE_WR |`
	`FS_FILE_ACCESS_MODE_CREATE |`
	`FS_FILE_ACCESS_MODE_TRUNC`

"`a`" or "`ab`"	`FS_FILE_ACCESS_MODE_WR |`
	`FS_FILE_ACCESS_MODE_CREATE |`
	`FS_FILE_ACCESS_MODE_APPEND`

"`r+`" or "`rb+`" or "`r+b`"	`FS_FILE_ACCESS_MODE_RD |` `FS_FILE_ACCESS_MODE_WR`

"`w+`" or "`wb+`" or "`w+b`"	`FS_FILE_ACCESS_MODE_RD |`
	`FS_FILE_ACCESS_MODE_WR |`
	`FS_FILE_ACCESS_MODE_CREATE |`
	`FS_FILE_ACCESS_MODE_TRUNC`

"`a+`" or "`ab+`" or "`a+b`"	`FS_FILE_ACCESS_MODE_RD |`
	`FS_FILE_ACCESS_MODE_WR |`
	`FS_FILE_ACCESS_MODE_CREATE |`
	`FS_FILE_ACCESS_MODE_APPEND`

Table 5-4. `fopen()` mode strings and `mode` equivalents.

5.01.02 Getting Information About a File

Detailed information about an open file, such as size and date/time stamps, can be obtained using the `FSFile_Query()` function:

```
FS_ENTRY_INFO  info;

FSFile_Query( p_file,  <-- file pointer
              &info,   <-- pointer to info structure
              &err);   <-- return error
```

The `FS_FILE_INFO` structure has the following members:

- `Attrib` contains the file attributes (see Section 5.02.01).
- `Size` is the size of the file, in octets.
- `DateTimeCreate` is the creation timestamp of the file.
- `DateTimeWr` is the last write (or modification) timestamp of the file.
- `BlkCnt` is the number of blocks allocated to the file. For a FAT file system, this is the number of clusters occupied by the file data.
- `BlkSize` is the size of each block allocated in octets. For a FAT file system, this is the size of a cluster.

`DateTimeCreate` and `DateTimeWr` are structures of type `FS_DATE_TIME`.

5.01.03 Configuring a File Buffer

The file module has functions to assign and flush a file buffer that are equivalents to POSIX API functions; the primary difference is the advantage of valuable return error codes to the application.

File Module Function	POSIX API Equivalent
`void FSFile_BufAssign (FS_FILE *p_file, void *p_buf, FS_FLAGS mode, CPU_SIZE_T size, FS_ERR *p_err);`	`int fs_setvbuf (FS_FILE *stream, char *buf, int mode, fs_size_t size);`
`void FSFile_BufFlush (FS_FILE *p_file, FS_ERR *p_err);`	`int fs_fflush (FS_FILE *stream);`

For more information about and an example of configuring a file buffer, see Section 4.03.03.

5.01.04 File Error Functions

The file module has functions get and clear a file's error status that are almost exact equivalents to POSIX API functions; the primary difference is the advantage of valuable return error codes to the application.

File Module Function	POSIX API Equivalent
`void FSFile_ClrErr (FS_FILE *p_file, FS_ERR *p_err);`	`void fs_clearerr (FS_FILE *stream);`
`CPU_BOOLEAN FSFile_IsErr (FS_FILE *p_file, FS_ERR *p_err);`	`int fs_ferror (FS_FILE *stream);`
`CPU_BOOLEAN FSFile_IsEOF (FS_FILE *p_file, FS_ERR *p_err);`	`int fs_feof (FS_FILE *stream);`

For more information about this functionality, see Section 4.03.04.

5.01.05 Atomic File Operations Using File Lock

The file module has functions lock files across several operations that are almost exact equivalents to POSIX API functions; the primary difference is the advantage of valuable return error codes to the application.

File Module Function	POSIX API Equivalent
`void FSFile_LockGet (FS_FILE *p_file, FS_ERR *p_err);`	`void fs_flockfile (FS_FILE *file);`
`void FSFile_LockAccept (FS_FILE *p_file, FS_ERR *p_err);`	`int fs_ftrylockfile (FS_FILE *file);`
`void FSFile_LockSet (FS_FILE *p_file, FS_ERR *p_err);`	`void fs_funlockfile (FS_FILE *file);`

For more information about and an example of using file locking, see Section 4.03.05.

5.02 Entry Access Functions

The entry access functions provide an API for performing single operations on file system entries (files and directories), such as copying, renaming or deleting. Each of these operations is atomic; consequently, in the absence of device access errors, either the operation will have completed or no change to the storage device will have been made upon function return.

One of these functions, `FSEntry_Query()`, obtains information about an entry (including the attributes, date/time stamp and file size). Two functions set entry properties, `FSEntry_AttribSet()` and `FSEntry_TimeSet()`, which set a file's attributes and date/time stamp. A new file entry can be created with `FSEntry_Create()` or an existing entry deleted, copied or renamed (with `FSEntry_Del()`, `FSEntry_Copy()` or `FSEntry_Rename()`).

Function	Description
`FSEntry_AttribSet()`	Set a file or directory's attributes.
`FSEntry_Copy()`	Copy a file.
`FSEntry_Create()`	Create a file or directory.
`FSEntry_Del()`	Delete a file or directory.
`FSEntry_Query()`	Get information about a file or directory.
`FSEntry_Rename()`	Rename a file or directory.
`FSEntry_TimeSet()`	Set a file or directory's date/time.

Table 5-2. Entry API Functions

5.02.01 File and Directory Attributes

The `FSEntry_Query()` function gets information about file system entry, including its attributes, which indicate whether it is a file or directory, writable or read-only, and visible or hidden:

```
FS_FLAGS        attrib;
FS_ENTRY_INFO  info;

FSEntry_Query( “path_name”,  <-- pointer to full path name
              &info,         <-- pointer to info
             &err);          <-- return error

attrib = info.Attrib;
```

The return value is a logical OR of attribute flags:

`FS_ENTRY_ATTRIB_RD`	Entry is readable.
`FS_ENTRY_ATTRIB_WR`	Entry is writable.
`FS_ENTRY_ATTRIB_HIDDEN`	Entry is hidden from user-level processes.
`FS_ENTRY_ATTRIB_DIR`	Entry is a directory.
`FS_ENTRY_ATTRIB_ROOT_DIR`	Entry is a root directory.

If no error is returned and `FS_ENTRY_ATTRIB_DIR` is not set, then the entry is a file.

An entry can be made read-only (or writable) or hidden (or visible) by setting its attributes:

```
attrib = FS_ENTRY_ATTRIB_RD;

FSEntry_AttribSet(“path_name”,      <-- pointer to full path name
                  attrib,           <-- attributes
                 &err);             <-- return error
```

The second argument should be the logical OR of relevant attribute flags.

`FS_ENTRY_ATTRIB_RD`	Entry is readable.
`FS_ENTRY_ATTRIB_WR`	Entry is writable.
`FS_ENTRY_ATTRIB_HIDDEN`	Entry is hidden from user-level processes.

If a flag is clear (not OR’d in), then that attribute will be clear. In the example above, the entry will be made read-only (i.e., not writable) and will be visible (i.e., not hidden) since the `WR` and `HIDDEN` flags are not set in `attrib`. Since there is no way to make files write-only (i.e., not readable), the `RD` flag should always be set.

5.02.02 File and Directory Times

The `FSEntry_Query()` function gets information about file system entry, including its timestamps:

```
FS_DATE_TIME   *p_date_time;
FS_ENTRY_INFO   info;

FSENTRY_Query("path_name",                  <-- pointer to full path name
              &info,                        <-- pointer to info
               &err);                       <-- return error

p_date_time = &info.DateTimeWr;
```

In addition to `DateTimeWr` which represents the modified date/time for the file or directory, created and accessed date/time information can be found in the same way by using `info.DateTimeCreate` and `info.DateAccess` fields.

The date/time of a file or directory can be set using the `FSEntry_TimeSet()` function:

```
FSEntry_TimeSet("path_name",                <-- pointer to full path name
                &date_time,                 <-- pointer to date/time structure
                 FS_DATE_TIME_MODIFY,       <-- flag to indicate Modified date/time set.
                &err);                      <-- return error
```

This function validates the ranges of the members of `date_time` (e.g., `date_time.Month` is checked to be between 0 and 11 inclusive); if `date_time` is erroneous, the error `FS_ERR_FILE_INVALID_DATE_TIME` is returned. However, the date/time itself is NOT validated, e.g., `date_time.Month` as 1 (February) and `date_time.Day` as 30 (30th) will not be rejected BUT specify an invalid date.

For more information about the `FS_DATE_TIME` structure, see Section C.03.

5.02.03 Creating New Files and Directories

A new file can be created using `FSFile_Open()` or `fs_fopen()`, if opened in write or append mode. There are a few other ways that new files can be created (most of which also apply to new directories).

The simplest is the `FSEntry_Create()` function, which just makes a new file or directory:

```
FSEntry_Create("\\file.txt",   <-- file name
               DEF_NO,         <-- DEF_NO means entry will be a file
               DEF_NO,         <-- DEF_NO means creation NOT exclusive
               &err);          <-- return error
```

If the second argument, `dir`, is `DEF_YES` the new entry will be a directory. The third argument, `excl`, indicates whether the creation should be exclusive. If it is exclusive (`excl` is `DEF_YES`), nothing will happen if the file already exists. Otherwise, the file currently specified by the file name will be deleted and a new empty file with that name created.

Similar functions exist to copy and rename an entry:

```
FSEntry_Copy("\\dir\\src.txt",    <-- source file name
             "\\dir\\dest.txt »,  <-- destination file name
             DEF_NO,              <-- DEF_NO means creation not exclusive
             &err);               <-- return error

FSEntry_Rename ("\\dir\\oldname.txt", <-- old file name
                "\\dir\\newname.txt", <-- new file name
                DEF_NO,               <-- DEF_NO means creation not exclusive
                &err);                <-- return error
```

(`FSEntry_Copy()` can only be used to copy files.) The first two arguments of each of these are both FULL paths; the second path is not relative to the parent directory of the first. As with `FSEntry_Create()`, the third argument of each, `excl`, indicates whether the creation should be exclusive. If it is exclusive (`excl` is `DEF_YES`), nothing will happen if the destination or new file already exists.

5.02.04 Deleting Files and Directories

A file or directory can be deleted using `FSEntry_Del()`:

```
FSEntry_Del("\\dir",    <-- entry name
            DEF_NO,     <-- DEF_NO means entry must be a dir
            &err);      <-- return error
```

The second argument, `file`, restricts deletion to directories. If it is `DEF_NO`, then the entry specified by the first argument MUST be a directory; if it is a file, an error will be returned. If it is `DEF_YES`, then the entry will be deleted whether it is a file or a directory.

Chapter

6

Directories

An application stores information in a file system by creating a file or appending new information to an existing file. At a later time, this information may be retrieved by reading the file. However, if a certain file must be found, or all files may be listed, the application can iterate through the entries in a directory using the **directory access (or simply directory) functions**. The available directory functions are listed in Table 6-1.

A separate set of **directory operations (or entry) functions** manage the files and directories available on the system. Using these functions, the application can create, delete and rename directories, and get and set a directory's attributes and date/time. More information about the entry functions can be found in Section 6.2.

The entry functions and the directory `Open()` function accept one or more **full directory paths**. For information about using file and path names, see Section 2.05.

The functions listed in Table 6-1 are core functions in the directory access module (`FSDir_####()` functions). These are matched by API level functions that correspond to standard C or POSIX functions. More information about the API-level functions can be found in Section 4. The core and API functions provide basically the same functionality; the benefits of the former are enhanced capabilities, a consistent interface and meaningful return error codes.

6.01 Directory Access Functions

The directory access functions provide an API for iterating through the entries within a directory. The `FSDir_Open()` function initiates this procedure, and each subsequent call to `FSDir_Rd()` (until all entries have been examined) returns a `FS_DIRENT` which holds information about a particular entry. The `FSDir_Close()` function releases any file system structures and locks.

Function	Description
`FSDir_Open()`	Open a directory.
`FSDir_Close()`	Close a directory
`FSDir_Rd()`	Read a directory entry.

Table 6-1. Directory API Functions

These functions are almost exact equivalents to POSIX API functions; the primary difference is the advantage of valuable return error codes to the application.

Directory Module Function	POSIX API Equivalent
`FS_DIR *FSDir_Open(CPU_CHAR *p_name_full, FS_ERR *p_err);`	`FS_DIR *fs_opendir (const char *dirname);`
`void FSDir_Close(FS_DIR *p_dir, FS_ERR *p_err);`	`int fs_closedir (FS_DIR *dirp);`
`void FSDir_Rd (FS_DIR *p_dir, FS_DIR_ENTRY *p_dir_entry FS_ERR *p_err);`	`int fs_readdir_r (FS_DIR *dirp, struct fs_dirent *entry, struct fs_dirent **result);`

For more information about and an example of using directories, see Section 4.04.

Chapter

7

File Systems: FAT

The FAT (File Allocation Table) file system, introduced as a simple file system for small disk drives, still predominates the removable storage market, because it is supported by all major operating systems. Since FAT's inception, it has been extended to support larger disks as well as longer file names. However, it remains simple enough for the most resource-constrained embedded system.

7.01 FAT Architecture

A FAT volume consists of four basic areas:

1. **Reserved area**. The reserved area includes the boot sector, which contains basic format information, like the number of sectors in the volume.
2. **FAT**. The FAT is a large table with one entry for each cluster. Each entry contains one of three values: the free cluster mark (indicating that it is not allocated), the cluster number of the next entry in the file (essentially, a link in a list of the file's clusters), or the end-of-cluster mark (indicating that it is the final cluster in the file).
3. **Root directory**. Note that in FAT32 volumes, the root directory is also part of the data area.
4. **Data area**. The data area contains files and directories, which are just a special type of file.

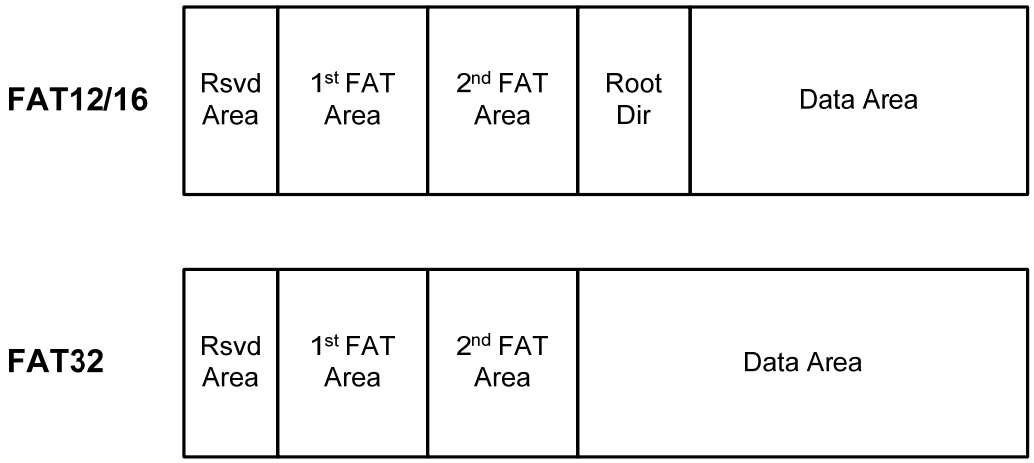


Figure 7-1. FAT Volume Layout.

7.01.01 FAT12 / FAT16 / FAT32

The earliest version of FAT, the file system integrated into Microsoft's DOS, was FAT12, so-called because each entry in the File Allocation Table was 12-bits. This limited disk size to approximately 32-MB. Extensions to 16- and 32-bit entries (FAT16 and FAT32) expand support to 2-GB and 8-TB, respectively. As described in Section D.07, support for FAT12, FAT16 and FAT32 can be individually disabled, if desired.

FAT32 introduces several new innovations above its predecessors. The root directory in the earlier systems was a fixed size; i.e., when the medium is formatted, the maximum number of files that could be created in the root directory (typically 512) is set. In FAT32, the root directory is dynamically resizable, like all other directories. Two special sectors are also included: the FS info sector and the backup boot sector. The former stores information convenient to the operation of the host, such as the last used cluster. The latter is a copy of the first disk sector (the boot sector), in case the original is corrupted.

7.01.02 Short and Long File Names

In the original version of FAT, files could only carry short "8 dot 3" names, with eight or fewer characters in the main name and three or fewer in its extension. The valid characters in these names are letters, digits, characters with values greater than 127 and the following:

```
$  %  '  -  _  @  ~  `  !  (  )  {  }  ^  #  &
```

In µC/FS, the name passed by the application is always verified, both for invalid length and invalid characters. If valid, the name is converted to upper case for storage in the directory entry.

Eventually, in a backwards-compatible extension, Microsoft introduced long file names (LFNs). LFNs are limited to 255 characters, stored as 16-bit Unicode in long directory entries. Each name is stored with a short file name composed by attaching a numeric "tail" to the original; this results in names like "file~1.txt". In addition to the characters allowed in SFNs, the following are allowed in LFNs:

```
+  ,  ;  =  [  ]
```

As described in Section D.07, support for LFNs can be disabled, if desired. If LFNs are enabled, the application may choose to specify file names in UTF-8 format, which will be converted to 16-bit Unicode for storage in directory entries. This option is available if `FS_CFG_UTF8_EN` is `DEF_ENABLED` (see Section D.02.06).

7.01.03 Directories and Directory Entries

In the FAT file system, directories are just special files, composed of 32-byte structures called **directory entries**. The topmost directory, the root directory, is located using information in the boot sector. The normal (short file name) entries in this directory and all other directories follow the format shown in Figure 7-2, with the following fields:

- **Name** is the 11-character 8.3 SFN.
- **Attr** are the attributes of the entry, indicating whether it is a file or directory, writable or read-only and visible or hidden.
- **Creation Time** and **Creation Date** are the time and date when the entry was created.
- **Access Date** is the date on which the file was last accessed.
- **Write Time** and **Write Date** are the time and date when the entry was last modified.
- **1st Cluster High** and **1st Cluster Low** contain the first cluster containing the file's data.
- **File Size** is the file size, in octets. If the entry is a directory, this is blank.

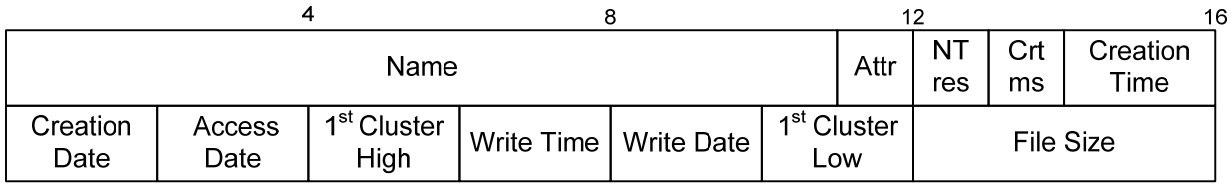


Figure 7-2. FAT Directory Entry (SFN Entry).

Within µC/FS, these are called Short File Name entries or **SFN entries**.

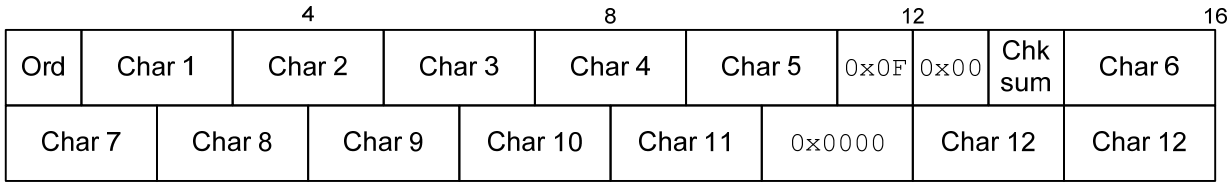


Figure 7-2. LFN Directory Entry.

To extend FAT for longer names, Microsoft devised the **LFN directory entry**, as shown in Figure 7-2. Thirteen characters overlay the fields in a traditional SFN entry, in addition to several important markers. The zeroth byte of the entry gives its order in the LFN entry sequence; the first always has the sixth bit set. If three entries were necessary, they would carry order numbers of `0x43`, `0x02` and `0x01`, respectively. None of these, you may note, are valid characters (which allows backward compatibility). Byte 11, where the attributes value is in a SFN, is always `0x0F`; Microsoft found that no older software would modify or use a directory entry with this marker. Figure 7-3 gives an example of the directory entries created for the file "abcdefghijklm.op". The checksum, stored in byte 13, is calculated from the SFN. It is checked each time the directory entries are parsed; if incorrect, the file system software knows that the SFN was modified (presumably by a system not LFN-aware).

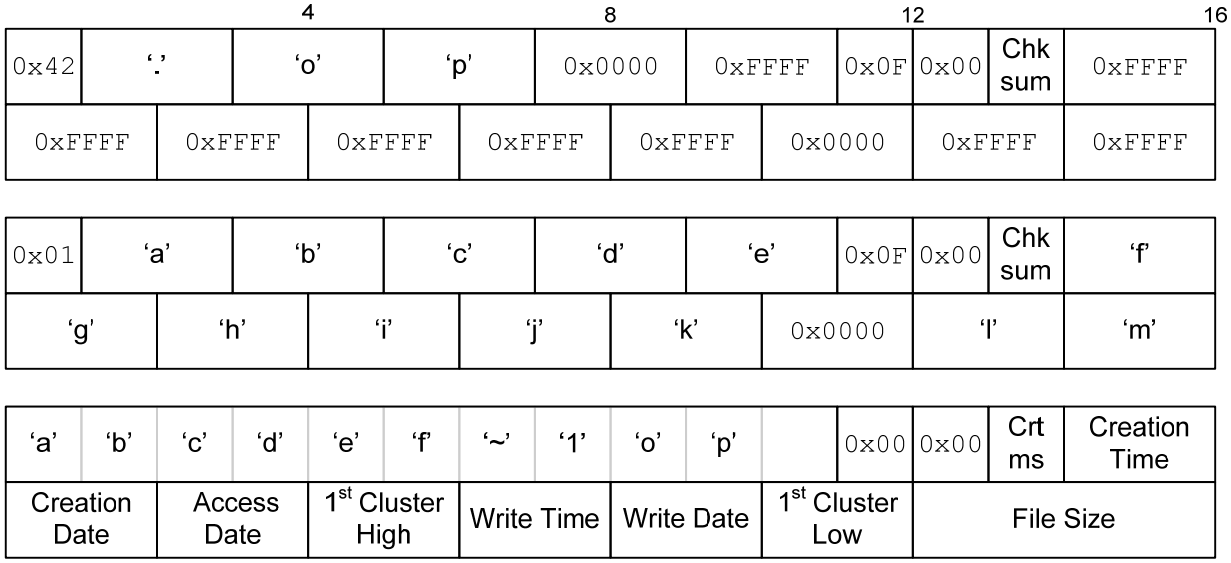


Figure 7-3. SNF entry and LFN entries for file named "abcdefghijklm.op".

7.01.04 FAT System Driver Architecture

As shown in Figure 2-2, the FAT system driver intermediates between functions that access files and directories (e.g., `fs_fopen()`) and volume read/write functions. Internally, the FAT system driver is divided into three subsystems, as shown in Figure 7-4. The first consists of the core functions directly called by file, directory, entry and volume modules. Next are the functions that understand the layout of the File Allocation Table and can allocate and free clusters. The final subsystem can create SFN and LFN directory entries and search a directory for a specific entry.

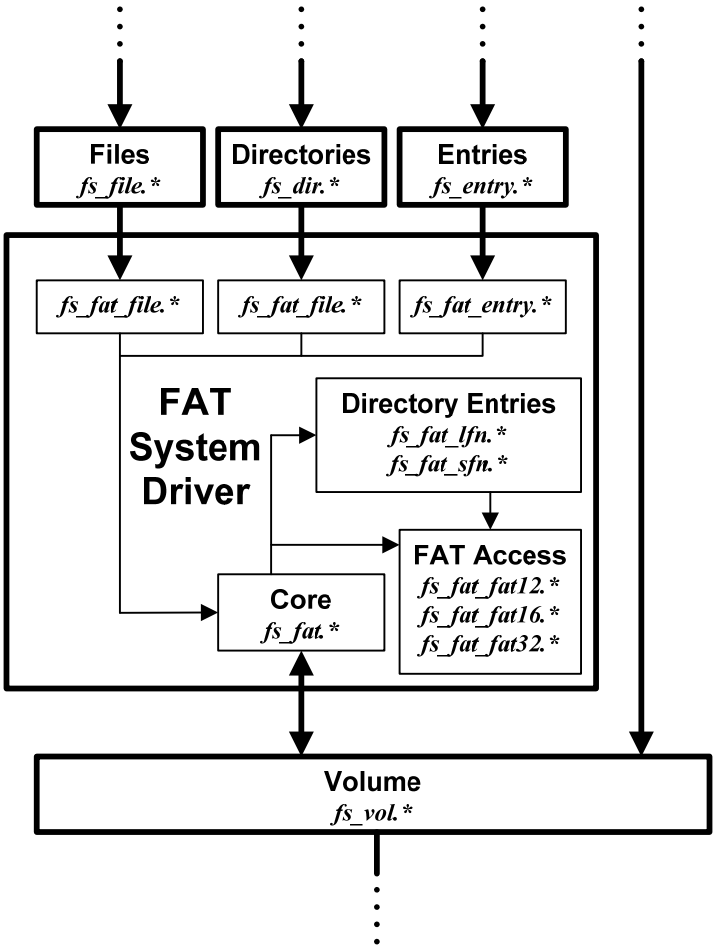


Figure 7-4. FAT system driver architecture.

7.02 Operations

The application rarely needs to know about the underlying file system; the FAT system driver within µC/FS handles file and volume accesses in a transparent manner. A few specific cases, the application may benefit from increased awareness of FAT operation.

7.02.01 Formatting

A volume, once it is open, may need to be formatted before files or directories can be created. The default format is selected by passing a NULL pointer as the second parameter of `FSVol_Fmt()`. Alternatively, the exact properties of the file system can be configured with a `FS_FAT_SYS_CFG` structure. An example of populating and using the FAT configuration is shown in Listing 7-1. If the configuration is invalid, an error will be returned from `FSVol_Fmt()`. For more information about the `FS_FAT_SYS_CFG` structure, see Section C.02.

Listing 7-1. Example device format.

```
void  App_InitFS (void)
{
    FS_ERR          err;
    FS_FAT_SYS_CFG  fat_cfg;

    .
    .
    .

    fat_cfg.ClusSize        =   4;                  /* Cluster size        = 4 * 512-B = 2-kB.*/
    fat_cfg.RsvdAreaSize    =   1;                  /* Reserved area       = 1 sector.        */
    fat_cfg.RootDirEntryCnt = 512;                  /* Entries in root dir = 512.             */
    fat_cfg.FAT_Type        =  12;                  /* FAT type            = FAT12.           */
    fat_cfg.NbrFATs         =   2;                  /* Number of FATs      = 2.               */

    FSVol_Fmt("ram:0:", &fat_cfg, &err);

    if (err != FS_ERR_NONE) {
        APP_TRACE_DEBUG(("Format failded.\r\n"));
    }

    .
    .
    .
}
```

7.02.02 Disk Check

Errors may accrue on a FAT volume, either by device removal during file system modifications or by improper host operation. Several corruptions are common:

- **Cross-linked files**. If a cluster becomes linked to two files, then it is called "cross-linked". The only way to resolve this is by deleting both files; if necessary, they can be copied first so that the contents can be verified.

- **Orphaned directory entries**. If LFNs are used, a single file name may span several directory entries. If a file deletion is interrupted, some of these may be left behind or "orphaned" to be deleted later.

- **Invalid cluster**. The cluster specified in a directory entry or linked in a chain may be invalid. The only recourse is to zero the cluster (if in a directory entry) or replace with end-of-cluster (if in a chain).

- **Chain length mismatch**. Too many or too few clusters may be linked to a file, compared to its size. If too many, the extra clusters should be freed. If too few, the file size should be adjusted.

- **Lost cluster**. A lost cluster is marked as allocated in the FAT, but is not linked to any file. Optionally, lost cluster chains may be recovered to a file.

7.02.03 Journaling

Since cluster allocation information is stored separately from file information and directory entries, most operations, such as adding data to a file, are non-atomic. The repercussions can be innocuous (e.g., wasted disk space) or serious (e.g., directory corruption). µC/FS includes an optional journaling add-on to its FAT system driver. System actions—such as creating a new file—are wrapped by updates to a special journal file. The journal is a compendium of **logs**, descriptions of the system before and after . When an operation is started, an **enter log** is added to the journal; upon completion, an **exit log** is added. Logs, for more complex operations, may be nested, the outer log giving additional context to the inner.

Listing 7-2. Opening the journal.

```
void  App_InitFS (void)
{
    FS_ERR  err;


    .
    .
    .

    /* Init the FS, Open Device and Volume(s) */


    FS_FAT_JournalOpen("sd:0:",
                       &err);
    APP_TEST_FAULT(err, FS_ERR_NONE);

    .
    .
    .

}
```

Listing 7-3 Starting and stopping journaling.

```
void  App_Task (void *p_arg)
{
    .
    .
    .

    FS_FAT_JournalStart("sd:0:", &err);            /* Start journaling.                    */

    .
    .                                              /* Perform fail-safe operations.        */
    .

    FS_FAT_JournalStop("sd:0:", &err);             /* Stop  journaling.                    */

    .
    .                                              /* Perform non-fail-safe operations.    */
    .
}
```

Chapter

8

Device Drivers

The file system initializes, controls, reads and writes a device using a **device driver**. A µC/FS device driver has eight interface functions, grouped into a `FS_DEV_DRV` structure that is registered with the file system (with `FS_DevDrvAdd()`) as part of application start-up, immediately following `FS_Init()`.

Several restrictions are enforced to preserve the uniqueness of device drivers and simplify management:

- Each device driver must have a unique name.
- No driver may be registered more than once.
- Device drivers cannot be unregistered.
- All device driver functions must be implemented (even if one or more is 'empty').

8.01 Provided Device Drivers

Portable device drivers are provided for standard media categories:

- **IDE driver**. The IDE driver supports compact flash (CF) cards and ATA IDE hard drives.
- **MSC driver**. The MSC (Mass Storage Class) driver supports USB host MSC devices (i.e., thumb drives or USB drives) via µC/USB-Host.
- **NOR driver**. The NOR flash driver support parallel (typically CFI-compliant) and serial (typically SPI) NOR flash devices.
- **RAM disk driver**. The RAM disk driver supports using internal or external RAM as a storage medium.
- **SD/MMC driver**. The SD/MMC driver supports SD, SD high-capacity and MMC cards, including micro and mini form factors. Either cardmode and SPI mode can be used.

Table 7-1 summarizes the drivers, driver names and driver API structure names. If you require more information about a driver, please consult the listed chapter.

Driver	Driver Name	Driver API Structure Name	Chapter
IDE/CF	"ide:"	`FSDev_IDE`	9
MSC	"msc:"	`FSDev_MSC`	11
NOR	"nor:"	`FSDev_NOR`	13
RAM disk	"ram:"	`FSDev_RAM`	14
SD/MMC	"sd:" / "sdcard:"	`FSDev_SD_SPI` / `FSDev_SD_Card`	15

Table 8-1. Device driver API structures.

If your medium is not supported by one of these drivers, a new driver can be written based on the template driver. Section 8.02 describes how to do this.

8.01.01 Driver Characterization

Typical ROM requirements are summarized in Table 8-2. The ROM data were collected on IAR EWARM v5.40.

Driver	ROM, Thumb Mode		ROM, ARM Mode	
	High Size Opt	High Speed Opt	High Size Opt	High Speed Opt
IDE/CF *	3.9 kB	4.5 kB	5.2 kB	5.9 kB
MSC**	0.7 kB	0.7 kB	1.1 kB	1.2 kB
NAND	---	---	---	---
NOR***	10.5 kB	12.2 kB	14.1 kB	16.3 kB
RAM disk	0.9 kB	1.0 kB	1.2 kB	1.4 kB
SD/MMC CardMode*	6.1 kB	7.3 kB	8.3 kB	10.1 kB
SD/MMC SPI*	5.3 kB	6.0 kB	7.4 kB	8.4 kB

Table 8-2. Driver ROM requirements.
* Not including BSP
**Not including µC/USB
***Not including physical-level driver or BSP

Typical RAM requirements are summarized in Table 8-3.

Driver	RAM (Overhead)	RAM (Per Device)
IDE/CF	8 bytes	24 bytes
MSC*	12 bytes	32 bytes
NAND	---	---
NOR***	8 bytes	--- bytes
RAM disk	8 bytes	24 bytes
SD/MMC CardMode	8 bytes	54 bytes
SD/MMC SPI	8 bytes	54 bytes

Table 8-3. Driver RAM requirements
*Not including µC/USB
***See Section 13.02

Performance can vary significantly as a result of CPU and hardware differences, both as well as file system format. Table 8-4 lists results for three general performance tests:

- **Read file test**. Read a file in 4-kB chunks. The time to open the file is NOT included in the time.
- **Write file test**. Write a file in 4-kB chunks. The time to open (create) the file is NOT included in the time.

Driver	CPU	Configuration	Performance (kB/s)	
			Read file	Write file
IDE/CF*	Freescale iMX27	200-MHz	7930 kB/s	1140 kB/s
MSC**	NXP LPC2468	48-MHz	309 kB/s	142 kB/s
NOR (parallel)***	ST STM32F103VE	72-MHz	1820 kB/s	213 kB/s
NOR (serial) §	ST STM32F103VE	72-MHz	691 kB/s	55 kB/s
RAM disk	NXP LPC2468	48-MHz	8260 kB/s	4530 kB/s
SD/MMC CardMode§§	NXP LPC2468	48-MHz, 1-bit mode	1010 kB/s	387 kB/s
SD/MMC CardMode§§	NXP LPC2468	48-MHz, 4-bit mode	2310 kB/s	557 kB/s
SD/MMC SPI§§	NXP LPC2468	48-MHz	405 kB/s	212 kB/s
SD/MMC SPI§§	NXP LPC2468	48-MHz (w/CRC)	356 kB/s	197 kB/s

Table 8-4. Driver performance (file test)
*Using 4-GB SanDisk Ultra II CF card
**Using 1-GB SanDisk Cruzer Micro
***Using ST M29W128GL NOR
§Using ST M25P64 serial flash
§§Using 2-GB SanDisk Ultra II SD card

8.02 Writing a Device Driver

A device driver is registered with the file system by passing a pointer to its API structure as the first parameter of `FS_DevDrvAdd()`. The API structure includes pointers to eight functions used to control and access the device.

Function	Description
`FSDev_####_NameGet()`	Get driver name.
`FSDev_####_Init()`	Initialize driver.
`FSDev_####_Open()`	Open a device.
`FSDev_####_Close()`	Close a device.
`FSDev_####_Rd()`	Read from a device.
`FSDev_####_Wr()`	Write to a device.
`FSDev_####_Query()`	Get information about a device.
`FSDev_####_IO_Ctrl()`	Execute device I/O control operation.

Table 8-5. Device Driver API Functions.

Listing 8-1. Device driver API.

```
const  FS_DEV_API  FSDev_#### = {
    FSDev_####_NameGet,
    FSDev_####_Init,
    FSDev_####_Open,
    FSDev_####_Close,
    FSDev_####_Rd,
#if (FS_CFG_RD_ONLY_EN == DEF_DISABLED)
    FSDev_####_Wr,
#endif
    FSDev_####_Query,
    FSDev_####_IO_Ctrl
};
```

8.02.01 Get Driver Name: `FSDev_####_NameGet()`

Device drivers are identified by unique names, on which device names are based. For example, the unique name for the Nor flash driver is “nor:”; the Nor devices will be named “nor:0:”, “nor:1:”, etc.

Prototype

```
static  const  CPU_CHAR  *FSDev_####_NameGet (void);
```

Arguments

None.

Returned Value

Pointer to the device driver name.

Notes/Warnings

1. The name MUST NOT include the ‘:’ character.
2. The name MUST be constant; each time this function is called, the same name MUST be returned.
3. The device driver `NameGet()` function is called while the caller holds the FS lock.

8.02.02 Initialize the Driver: `FSDev_####_Init()`

The device driver `Init()` function should initialize any structures, tables or variables that are common to all devices or are used to manage devices accessed with the driver. This function SHOULD NOT initialize any devices; that will be done individually for each with the device driver's `Open()` function.

Prototype

```
static  void  FSDev_####_Init (void);
```

Arguments

None.

Returned Value

None.

Notes/Warnings

1. The device driver `Init()` function is called while the caller holds the FS lock.

8.02.03 Open (Initialize) a Device Instance: FSDev_####_Open()

The device driver Open() function should initialize the hardware to access a device and attempt to initialize that device. If this function is successful (i.e., it returns FS_ERR_NONE), then the file system suite expects the device to be ready for read and write accesses.

Prototype

```
static  void  FSDev_####_Open (FS_DEV  *p_dev,
                               void    *p_dev_cfg,
                               FS_ERR  *p_err);
```

Arguments

p_dev Pointer to device to open.

p_dev_cfg Pointer to device configuration.

p_err Pointer to variable that will receive the return error code from this function:

FS_ERR_NONE	Device opened successfully.
FS_ERR_DEV_ALREADY_OPEN	Device unit is already opened.
FS_ERR_DEV_INVALID_CFG	Device configuration specified invalid.
FS_ERR_DEV_INVALID_LOW_FMT	Device needs to be low-level formatted.
FS_ERR_DEV_INVALID_LOW_PARAMS	Device low-level device parameters invalid.
FS_ERR_DEV_INVALID_UNIT_NBR	Device unit number is invalid.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NOT_PRESENT	Device unit is not present.
FS_ERR_DEV_TIMEOUT	Device timeout.
FS_ERR_MEM_ALLOC	Memory could not be allocated.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should NEVER be called when a device is already open.

2. Some drivers may need to track whether a device has been previously opened (indicating that the hardware has previously been initialized).

3. This will be called EVERY time the device is opened.

4. The driver should identify the device instance to be opened by checking p_dev->UnitNbr. For example, if "template:2:" is to be opened, then p_dev->UnitNbr will hold the integer 2.

5. The device driver Open() function is called while the caller holds the device lock.

8.02.04 Close (Uninitialize) a Device Instance: FSDev_####_Close()

The device driver `Close()` function should uninitialize the hardware and release access to any resources acquired in the `Open()` function.

Prototype

```
static  void  FSDev_####_Close (FS_DEV  *p_dev);
```

Arguments

`p_dev` Pointer to device to close.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should ONLY be called when a device is open.

2. This will be called EVERY time the device is closed.

3. The device driver `Close()` function is called while the caller holds the device lock.

8.02.05 Read from a Device: FSDev_####_Rd()

The device driver Rd() function should read from a device and store data in a buffer. If an error is returned, the file system suite assumes that no data is read; if not all data can be read, an error MUST be returned.

Prototype

```
static  void  FSDev_####_Rd (FS_DEV      *p_dev,
                             void        *p_dest,
                             FS_SEC_NBR   start,
                             FS_SEC_QTY   cnt,
                             FS_ERR      *p_err);
```

Arguments

p_dev Pointer to device to read from.

p_dest Pointer to destination buffer.

start Start sector of read.

cnt Number of sectors to read

p_err Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Sector(s) read.
FS_ERR_DEV_INVALID_UNIT_NBR	Device unit number is invalid.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NOT_OPEN	Device is not open.
FS_ERR_DEV_NOT_PRESENT	Device is not present.
FS_ERR_DEV_TIMEOUT	Device timeout.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should ONLY be called when a device is open.

2. The device driver Rd() function is called while the caller holds the device lock.

8.02.06 Write to a Device: FSDev_####_Wr()

The device driver Wr() function should write to a device the data from a buffer. If an error is returned, the file system suite assumes that no data has been written.

Prototype

```
static  void  FSDev_####_Wr (FS_DEV      *p_dev,
                             void        *p_src,
                             FS_SEC_NBR   start,
                             FS_SEC_QTY   cnt,
                             FS_ERR      *p_err);
```

Arguments

p_dev Pointer to device to write to.

p_src Pointer to source buffer.

start Start sector of write.

cnt Number of sectors to write

p_err Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Sector(s) written.
FS_ERR_DEV_INVALID_UNIT_NBR	Device unit number is invalid.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NOT_OPEN	Device is not open.
FS_ERR_DEV_NOT_PRESENT	Device is not present.
FS_ERR_DEV_TIMEOUT	Device timeout.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should ONLY be called when a device is open.

2. The device driver Wr() function is called while the caller holds the device lock.

8.02.07 Get Information about a Device: FSDev_####_Query()

The device driver `Query()` function gets information about a device.

Prototype

```
static  void  FSDev_####_Query (FS_DEV       *p_dev,
                                FS_DEV_INFO  *p_info,
                                FS_ERR       *p_err);
```

Arguments

`p_dev` Pointer to device to query.

`p_info` Pointer to structure that will receive device information.

`p_err` Pointer to variable that will receive the return error code from this function

`FS_ERR_NONE`	Device information obtained.
`FS_ERR_DEV_INVALID_UNIT_NBR`	Device unit number is invalid.
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should ONLY be called when a device is open.

2. The device driver `Query()` function is called while the caller holds the device lock.

For more information about the `FS_DEV_INFO` structure, see Section C.04.

8.02.08 Perform I/O Control Operation: FSDev_####_IO_Ctrl()

The device driver IO_Ctrl() function performs an I/O control operation.

Prototype

```
static  void  FSDev_####_IO_Ctrl (FS_DEV          *p_dev,
                                  FS_IO_CTRL_CMD   cmd,
                                  Void            *p_buf,
                                  FS_ERR          *p_err);
```

Arguments

p_dev Pointer to device to query.

p_buf Buffer which holds data to be used for operations
OR
Buffer in which data will be stored as a result of operation.

p_err Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Control operation performed successfully.
FS_ERR_DEV_INVALID_IO_CTRL	I/O control operation unknown to driver.
FS_ERR_DEV_INVALID_UNIT_NBR	Device unit number is invalid.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NOT_OPEN	Device is not open.
FS_ERR_DEV_NOT_PRESENT	Device is not present.
FS_ERR_DEV_TIMEOUT	Device timeout.

Returned Value

None.

Notes/Warnings

1. Tracking whether a device is open is not necessary, because this should ONLY be called when a device is open.

2. Defined I/O control operations are

a.	FS_DEV_IO_CTRL_REFRESH	Refresh device.
b.	FS_DEV_IO_CTRL_LOW_FMT	Low-level format device.
c.	FS_DEV_IO_CTRL_LOW_MOUNT	Low-level mount device.
d.	FS_DEV_IO_CTRL_LOW_UNMOUNT	Low-level unmount device.
e.	FS_DEV_IO_CTRL_LOW_COMPACT	Low-level compact device.
f.	FS_DEV_IO_CTRL_LOW_DEFRAH	Low-level defragment device.
g.	FS_DEV_IO_CTRL_SEC_RELEASE	Release data in sector
h.	FS_DEV_IO_CTRL_PHY_RD	Read physical device
i.	FS_DEV_IO_CTRL_PHY_WR	Write physical device
j.	FS_DEV_IO_CTRL_PHY_RD_PAGE	Read physical device page

k. `FS_DEV_IO_CTRL_PHY_WR_PAGE` Write physical device page
l. `FS_DEV_IO_CTRL_PHY_ERASE_BLK` Erase physical device block
m. `FS_DEV_IO_CTRL_PHY_ERASE_CHIP` Erase physical device

Not all of these operations are valid for all devices.

3. The device driver `IO_Ctrl()` function is called while the caller holds the device lock.

8.03 Writing a Driver SPI BSP

Device drivers which access the memory device though an SPI bus require an SPI BSP, with ten functions placed into an API structure. If several devices are located on the same bus, one of two approaches may be used:

- **Unique unit numbers**. All devices can use the same SPI BSP if and only if each device must has a unique unit number, which is the first argument of each BSP function. Only in this case will the unit number-to-chip select translation be a one-to-one mapping. For example, the SD/MMC card "sd:0:" (on chip select 1), and serial NOR "nor:1:" (on chip select 3) require only one BSP.

- **Unique SPI BSPs**. Device of more than one type (e.g., a SD/MMC card and a serial NOR) can have the same unit number if and only if each driver uses a separate BSP. For example, the SD/MMC card "sd:0:" (on chip select 1), and serial NOR "nor:0:" (on chip select 3) require two BSPs, one for the SD/MMC driver and another for the NOR driver.

Function	Description
`FSDev_####_BSP_SPI_Open()`	Open (initialize) SPI.
`FSDev_####_BSP_SPI_Close()`	Close (uninitialize) SPI.
`FSDev_####_BSP_SPI_Lock()`	Acquire SPI lock.
`FSDev_####_BSP_SPI_Unlock()`	Release SPI lock
`FSDev_####_BSP_SPI_Rd()`	Read from SPI.
`FSDev_####_BSP_SPI_Wr()`	Write to SPI.
`FSDev_####_BSP_SPI_ChipSelEn()`	Enable chip select.
`FSDev_####_BSP_SPI_ChipSelDis()`	Disable chip select.
`FSDev_####_BSP_SPI_SetClkFreq()`	Set SPI clock frequency.

Table 8-6. Device Driver API Functions.

Listing 8-2. Device SPI BSP API.

```
const  FS_DEV_SPI_API  FSDev_####_BSP_SPI = {
    FSDev_####_BSP_SPI_Open,
    FSDev_####_BSP_SPI_Close,
    FSDev_####_BSP_SPI_Lock,
    FSDev_####_BSP_SPI_Unlock,
    FSDev_####_BSP_SPI_Rd,
    FSDev_####_BSP_SPI_Wr,
    FSDev_####_BSP_SPI_ChipSelEn,
    FSDev_####_BSP_SPI_ChipSelDis,
    FSDev_####_BSP_SPI_SetClkFreq
};
```

8.03.01 SPI Communication

SPI is a simple protocol supported by peripherals commonly built-in on CPUs. Moreover, since the communication can easily be accomplished by software control of GPIO pins ("software SPI" or "bit-banging"), a SPI device can be connected to almost any platform. Any SPI device uses four signals are used to communicate with the host (CS, DataIn, CLK and DataOut). Some card holders contain circuitry for card detect and write protect indicators, which the MCU/MPU may also monitor.

Signal Name	Type	Description
CS	I	Chip select
MOSI	I	Master out, slave in
CLK	I	Clock
MISO	O	Master in, slave out

Table 8-7. SPI pins

The four signals connecting the host (or master) and card (also known as the slave) are named variously in different manuals and documents. The MOSI pin (Master Out Slave In) may be called DI on device pinouts; similarly, MISO pin (Master In Slave Out) may be calld DO on device pinouts. The CS and CLK pins (also known as SSEL and SCK) are the chip select and clock pins. The host selects the slave by asserting CS, potentially allowing it to choose a single peripheral among several that are sharing the bus (i.e., by sharing the CLK, MOSI and MISO signals).

8.03.02 Open (Initialize) SPI: `FSDev_####_BSP_SPI_Open()`

Open (initialize) hardware for SPI.

Prototype

```
CPU_BOOLEAN  FSDev_####_BSP_SPI_Open (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of device.

Returned Value

`DEF_OK`, if interface was opened.
`DEF_FAIL`, otherwise.

Notes/Warnings

1. This function will be called every time the device is opened.

2. Several aspects of SPI communication may need to be configured:

    a. **Clock polarity and phase (CPOL and CPHA).** SPI communication takes place in any of four modes depending on the clock phase and clock polarity settings:

        i. If CPOL = 0, the clock is low when inactive.
        If CPOL = 1, the clock is high when inactive.

        ii. If CPHA = 0, data is read on the leadng edge of the clock and changed on the following edge of the clock.
        If CPHA = 1, data is changed on the leading edge of the clock and read on the following edge of the clock.

    The driver should configure the SPI for CPOL = 0, CPHA = 0. This means that data (i.e., bits) are read when a low→high clock transition occurs and changed when a high→low clock transition occurs.

    b. **Transfer unit length**. A transfer unit is always 8-bits.

    c. **Shift direction**. The most significant bit should always be transmitted first.

3. The CS (Chip Select) MUST be configured as a GPIO output; it should not be controlled by the CPU's SPI peripheral. `FSDev_####_BSP_SPI_ChipSelEn()` and `FSDev_####_BSP_SPI_ChipSelDis()` manually enable and disable the CS.

8.03.03 Close (uninitialize) SPI: `FSDev_####_BSP_SPI_Close()`

Close (uninitialize) hardware for SD/MMC SPI.

Prototype

```
void  FSDev_####_BSP_SPI_Close (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of device.

Returned Value

None.

Notes/Warnings

This function will be called every time the device is closed.

8.03.04 Acquire/Release SPI Lock: `FSDev_####_BSP_SPI_Lock()/Unlock()`

Acquire/release SPI lock.

Prototypes

```
void  FSDev_####_BSP_SPI_Lock   (FS_QTY  unit_nbr);

void  FSDev_####_BSP_SPI_Unlock (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of device.

Returned Value

None.

Notes/Warnings

`FSDev_####_BSP_SPI_Lock()` will be called before the driver begins to access the SPI. The application should NOT use the same bus to access another device until the matching call to `FSDev_####_BSP_SPI_Unlock()` has been made.

8.03.05 Read Data from SD/MMC SPI: `FSDev_####_BSP_SPI_Rd()`

Read from device SPI bus.

Prototype

```
void  FSDev_####_BSP_SPI_Rd (FS_QTY       unit_nbr,
                             void        *p_dest,
                             CPU_SIZE_T   cnt);
```

Arguments

`unit_nbr` Unit number of device.

`p_dest` Pointer to destination buffer.

`cnt` Number of bytes to read from the SPI.

Returned Value

None.

Notes/Warnings

The data out (MOSI) line during data read often MUST be high (i.e., `0xFF` bytes MUST be transmitted). This is an issue since, in the SPI protocol, the controller is writing even while reading.

The chip select should NOT be controlled by a hardware SPI controller; rather, it should be controlled by `FSDev_####_BSP_SPI_ChipSelDis()` and `FSDev_####_BSP_SPI_ChipSelEn()`.

8.03.06 Write Data to SD/MMC SPI: `FSDev_####_BSP_SPI_Wr()`

Write to device SPI bus.

Prototype

```
void  FSDev_####_BSP_SPI_Wr (FS_QTY       unit_nbr,
                             void        *p_src,
                             CPU_SIZE_T   cnt);
```

Arguments

`unit_nbr` Unit number of device.

`p_src` Pointer to source buffer.

`cnt` Number of bytes to write to the SPI.

Returned Value

None.

Notes/Warnings

The chip select should NOT be controlled by a hardware SPI controller; rather, it should be controlled by `FSDev_####_BSP_SPI_ChipSelDis()` and `FSDev_####_BSP_SPI_ChipSelEn()`.

8.03.07 Enable Chip Select: `FSDev_####_BSP_SPI_ChipSelEn()`

Enable chip select for device.

Prototype

```
void  FSDev_####_BSP_SPI_ChipSelEn (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of device.

Returned Value

None.

Notes/Warnings

The chip select is typically “active low”; to enable the device, the chip select pin should be cleared.

8.03.08 Disable Chip Select: `FSDev_####_BSP_SPI_ChipSelDis()`

Disable chip select for device.

Prototype

```
void  FSDev_####_BSP_SPI_ChipSelDis (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of device.

Returned Value

None.

Notes/Warnings

The chip select is typically “active low”; to disable the device, the chip select pin should be set.

8.03.09 Set SPI Clock Frequency: `FSDev_####_BSP_SPI_SetClkFreq()`

Set SPI clock frequency.

Prototype

```
void  FSDev_####_BSP_SPI_SetClkFreq (FS_QTY      unit_nbr,
                                     CPU_INT32U  freq);
```

Arguments

`unit_nbr` Unit number of device.

`clk_freq` Clock frequency, in Hz

Returned Value

None.

Notes/Warnings

The effective clock frequency MUST be no more than `freq`. If the frequency cannot be configured equal to `freq`, it should be configured less than f`req`.

Chapter

9

IDE/CF Driver

Compact flash (CF) cards are portable, low-cost media often used for storage in consumer devices. Several variants, in different media widths, are widely available, all supported by the IDE driver. ATA IDE hard drives are also supported by this driver.

9.01 Files and Directories

The files inside the IDE driver directory are outlined in this section; the generic file system files, outlined in Section 1.02, are also required.

\Micrium\Software\uC-FS\Dev

This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\IDE

This directory contains the IDE driver files.

fs_dev_ide.* are device driver for IDE devices. This file requires a set of BSP functions be defined in a file named ***fs_dev_ide_bsp.c*** to work with a certain hardware setup.

.*BSP\Template\fs_dev_ide_bsp.c* is a template BSP. See Section 9.03 for more information.

\Micrium\Software\uC-FS\Examples\BSP\Dev\IDE

Each subdirectory contains an example BSP for a particular platform. These are named according to the following rubric:

<Chip Manufacturer>\<Board or CPU>\fs_dev_ide_bsp.c

9.02 Using the IDE/CF Driver

To use the IDE/CF driver, five files, in addition to the generic file system files, must be included in the build:

- *fs_dev_ide.c* (located in the directory specified in Section 9.01).
- *fs_dev_ide.h* (located in the directory specified in Section 9.01).
- *fs_dev_ide_bsp.c* (located in the user application or BSP).

The file *fs_dev_ide.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directories must be on the project include path:

- *\Micrium\Software\uC-FS\Drivers\IDE*

A single IDE/CF volume is opened as shown in Listing 9-1. The file system initialization (`FS_Init()`) function must have been previously called.

ROM/RAM characteristics and performance benchmarks of the IDE driver can be found in Section 8.01.01.

Listing 9-1. Opening a IDE/CF device volume.

```
CPU_BOOLEAN  App_FS_AddIDE (void)
{
    FS_ERR       err;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_IDE,           /* (1)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) && (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        return (DEF_FAIL);
    }
                                                     /* (2)    */
    FSDev_Open((CPU_CHAR *)"ide:0:",                 /*    (a) */
               (void     *) 0,                       /*    (b) */
               (FS_ERR   *)&err);


    switch (err) {
        case FS_ERR_NONE:
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
             return (DEF_FAIL);

        default:
             return (DEF_FAIL);
    }
                                                     /* (3)    */
    FSVol_Open((CPU_CHAR         *)"ide:0:",         /*    (a) */
               (CPU_CHAR         *)"ide:0:",         /*    (b) */
               (FS_PARTITION_NBR  ) 0,               /*    (c) */
               (FS_ERR           *)&err);

    switch (err) {
        case FS_ERR_NONE:
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
```

```
        case FS_ERR_PARTITION_NOT_FOUND:            /* (4)    */
             return (DEF_FAIL);

        default:
             return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

L9-1(1) Register the IDE/CF device driver.

L9-1(2) `FSDev_Open()` opens/initializes a file system device. The parameters are the device name (1a) and a pointer to a device driver-specific configuration structure (1b). The device name (1a) is composed of a device driver name ("ide"), a single colon, an ASCII-formatted integer (the unit number) and another colon. Since the IDE/CF driver requires no configuration, the configuration structure (1b) should be passed a NULL pointer.

Since IDE/CF are often removable media, it is possible for the device to not be present when `FSDev_Open()` is called. The device will still be added to the file system and a volume opened on the (not yet present) device. When the volume is later accessed, the file system will attempt to refresh the device information and detect a file system (see Section 3.01.02 for more information).

L9-1(3) `FSVol_Open()` opens/mounts a volume. The parameters are the volume name (2a), the device name (2b) and the partition that will be opened (2c). There is no restriction on the volume name (2a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partition, then the partition number (2c) should be zero.

L9-1(4) High level format can be applied to the volume if `FS_ERR_PARTITION_NOT_FOUND` is returned by the call to `FSVol_Open ()` function.

If the IDE initialization succeeds, the file system will produce the trace output as shown in Figure 9-2 (if a sufficiently high trace level is configured). See Section D.09 about configuring the trace level.

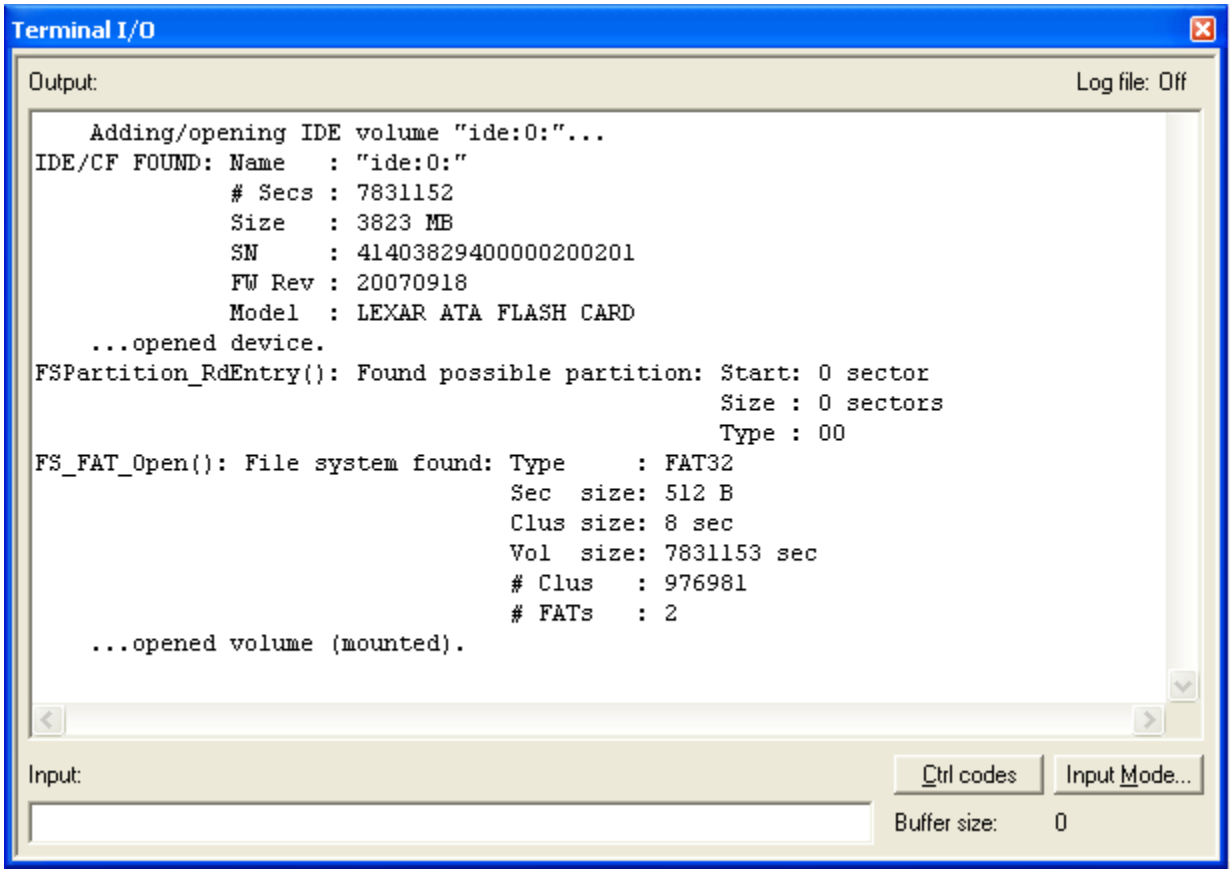


Figure 9-2. IDE Detection Trace Output.

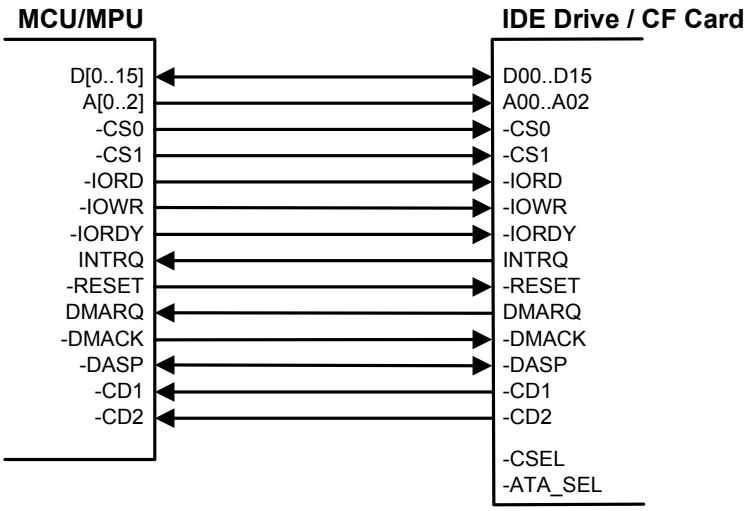
9.02.01 ATA (True IDE) Communication

The interface between an ATA device and host is comprised of data bus, address bus and various control signals, as shown in Figure 9-3. Three forms of data transfer are possible, each with several timing modes:

1. **PIO (programmed input/output)**. PIO must always be possible; indeed, it may be the only possible transfer form on certain hardware. Using PIO, data requests are satisfied by direct reads or writes to the DATA register. The IDENTIFY_DEVICE command and standard sector and multiple sector read/write commands always involve this type of transfer. Five timing modes (0, 1, 2, 3 and 4) are standard; two more (5 and 6) are defined in the CF specification.

2. **Mutiword DMA**. In Multiword DMA mode, a DMARQ and –DMACK handshake initiates automatic data transmission, during which the host moves data between its memory and the bus. The DMA read/write commands (READ_DMA, WRITE_DMA) may use Multiword DMA. Three timing modes (0, 1 and 2) are standard; two more (3 and 4) are defined in the CF specification.

3. **Ultra DMA**. The purposes of several control signals are reassigned during Ultra DMA transfers. For example, IORDY becomes either DDMARDY or DSTROBE (depending on the direction) to control data flow. The DMA read/write commands (READ_DMA, WRITE_DMA) may use Ultra DMA. Seven timing modes (0, 1, 2, 3, 4, 5 and 6) are standard.



Pin Name(s)	Function
A00, A01, A02, -CS0, -CS1	Address group. Use by host to select the register or data port that will be accessed.
-IORD	Asserted by host to read register or data port.
-IOWR	Asserted by host to write register or data port.
-IORDY	
INTRQ	Interrupt request to the host.
-RESET	Hardware reset signal.
DMARQ	Asserted by device when it is ready for a DMA transfer.
-DMACK	DMA acknowledge signal asserted by host in response to DMARQ.
-DASP	Disk Active/Slav Present signal in Master/Slave handshake protocol.
-CD1, -CD2	Chip detect.

Figure 9-3. True IDE (ATA) host/device connection.

The host controls the device via 8 registers (see Figure 9-4). Seven of these registers comprise the command block: FR, SC, SN, CYL, CYH, DH and CMD. The command block registers are written, in sequence, to execute a command. Afterwards, the error and status register return to the host a failure indicator or otherwise signal device operation completion. The need to poll these registers is removed if the host is instead alerted by an interrupt request (on the INTRQ signal) to attend to the device.

Up to two devices, known as master and slave (or device 0 and device 1) may be located on a single conventional bus. The active device (the target for the next command) is selected by the DEV bit in the DH register, and generally only one device can be accessed at a time, meaning that a read or write to one cannot interrupt a read or write to the other.

Abbreviation	Name	R/W	Control Signals				
			CS1	CS0	A02	A01	A00
DATA	Data	R/W	0	1	0	0	0
ERR	Error	R	0	1	0	0	1
FR	Features	W	0	1	0	0	1
SC	Sector Count	W	0	1	0	1	0
SN	Sector Number	W	0	1	0	1	1
CYL	Cylinder Low	W	0	1	1	0	0
CYH	Cylinder High	W	0	1	1	0	1
DH	Card/Drive/Head	W	0	1	1	1	0
CMD	Command	W	0	1	1	1	1
STATUS	Status	R	0	1	1	1	1
ALTSTATUS	Alternate Status	R	1	0	1	1	0
DEVCTRL	Device Control	W	1	0	1	1	0

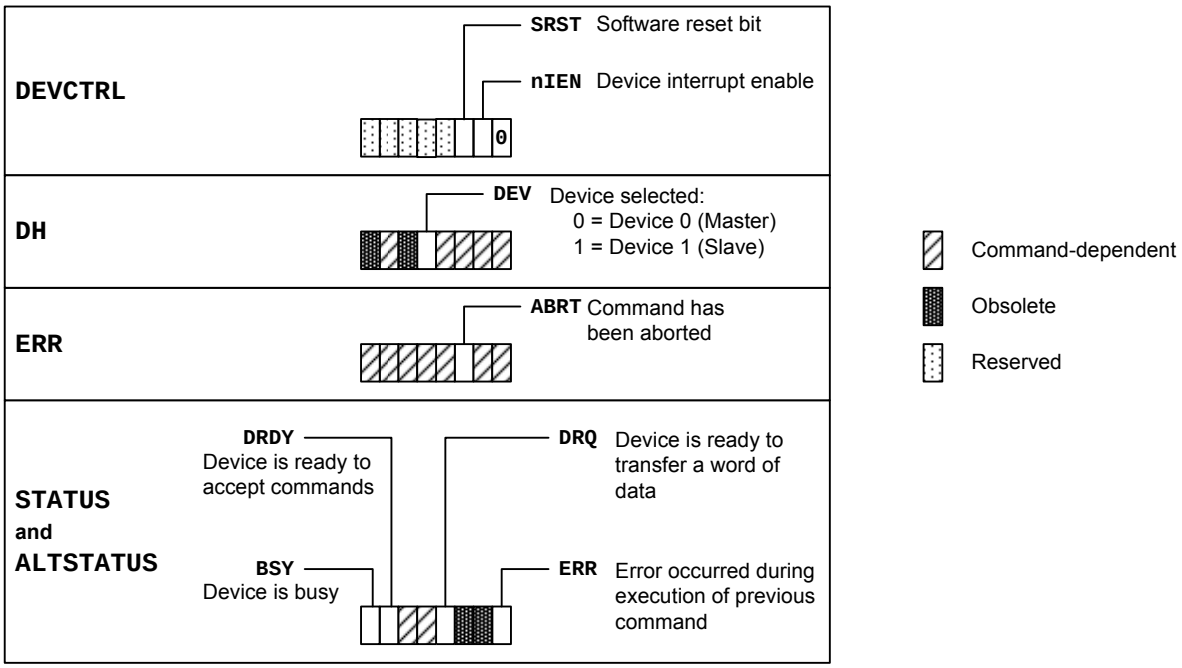


Figure 9-4. Register definitions.

9.02.02 IDE BSP Overview

A BSP is required so that the IDE driver will work on a particular system. The functions shown in the table below must be implemented.

Function	Description
`FSDev_IDE_BSP_Open()`	Open (initialize) hardware.
`FSDev_IDE_BSP_Close()`	Close (uninitialize) hardware.
`FSDev_IDE_BSP_Lock()`	Acquire IDE bus lock.
`FSDev_IDE_BSP_Unlock()`	Release IDE bus lock.
`FSDev_IDE_BSP_Reset()`	Hardware-reset IDE device
`FSDev_IDE_BSP_RegRd()`	Read from IDE device register.
`FSDev_IDE_BSP_RegWr()`	Write to IDE device register.
`FSDev_IDE_BSP_CmdWr()`	Write command to IDE device register.
`FSDev_IDE_BSP_DataRd()`	Read data from IDE device.
`FSDev_IDE_BSP_DataWr()`	Write data to IDE device.
`FSDev_IDE_BSP_DMA_Start()`	Setup DMA for command (Initialize channel).
`FSDev_IDE_BSP_DMA_End()`	End DMA transfer (and uninitialize channel).
`FSDev_IDE_BSP_GetDrvNbr()`	Get IDE drive number.
`FSDev_IDE_BSP_GetModesSupported()`	Get supported transfer modes.
`FSDev_IDE_BSP_SetMode()`	Set transfer modes.
`FSDev_IDE_BSP_Dly400_ns()`	Delay for 400 ns.

Table 9-1. IDE BSP Functions

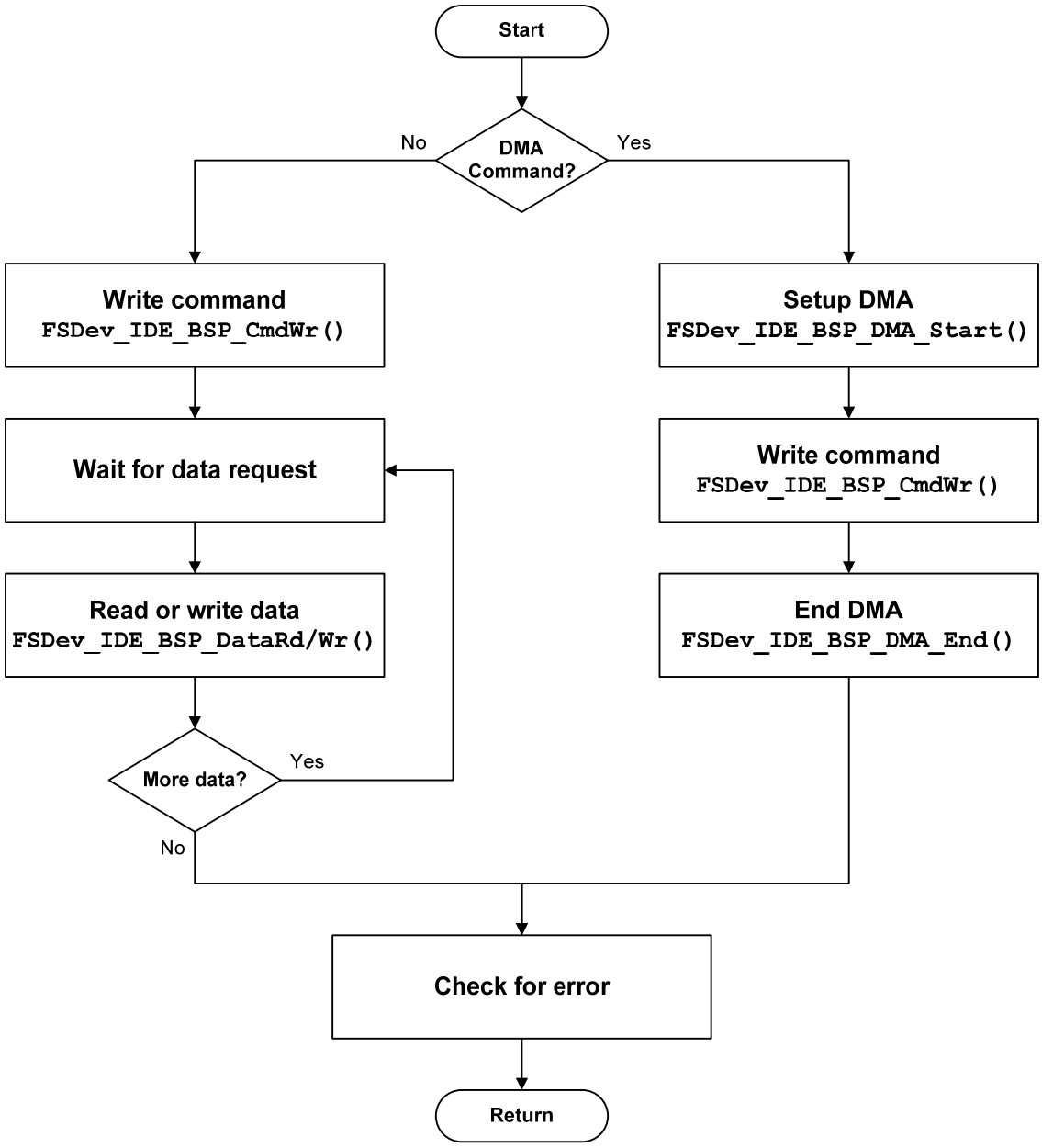


Figure 9-5. Command execution.

9.02.02.01 Open (Initialize) IDE Device: FSDev_IDE_BSP_Open( )

Each time the IDE device is opened with the file system, this function will be called to initialize the hardware.

Prototype

```
CPU_BOOLEAN  FSDev_IDE_BSP_Open (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

`DEF_OK`, if interface was opened.
`DEF_FAIL`, otherwise.

Notes/Warnings

This function will be called every time the IDE device is opened.

9.02.02.02 Close (Uninitialize) IDE Device: `FSDev_IDE_BSP_Close()`

Each time the IDE device is closed with the file system, this function will be called uninitialize the hardware.

Prototype

```
void  FSDev_IDE_BSP_Close (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

None.

Notes/Warnings

This function will be called every time the IDE device is closed.

9.02.02.03 Acquire / Release IDE Bus Lock: FSDev_IDE_BSP_Lock()/Unlock()

Acquire/release IDE bus lock.

Prototypes

```
void  FSDev_IDE_BSP_Lock   (FS_QTY  unit_nbr);

void  FSDev_IDE_BSP_Unlock (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

None.

Notes/Warnings

`FSDev_IDE_BSP_Lock()` will be called before the IDE driver begins to access the IDE bus. The application should NOT use the same bus to access another device until the matching call to `FSDev_IDE_BSP_Unlock()` has been made.

9.02.02.04 Reset IDE Device: FSDev_IDE_BSP_Reset()

Hardware-reset the IDE device.

Prototype

```
void  FSDev_IDE_BSP_Reset (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

None.

Notes/Warnings

None.

9.02.02.05 Read Register on IDE Device: FSDev_IDE_BSP_RegRd()

Read from IDE device register.

Prototype

```
CPU_INT08U  FSDev_IDE_BSP_RegRd (FS_QTY      unit_nbr,
                                 CPU_INT08U  reg);
```

Arguments

`unit_nbr` Unit number of IDE device.

`reg` Register to read :

`FS_DEV_IDE_REG_ERR`	Error Register.
`FS_DEV_IDE_REG_SC`	Sector Count Register.
`FS_DEV_IDE_REG_SN`	Sector Number Register.
`FS_DEV_IDE_REG_CYL`	Cylinder Low Register.
`FS_DEV_IDE_REG_CYH`	Cylinder High Register.
`FS_DEV_IDE_REG_DH`	Card/Drive/Head Register.
`FS_DEV_IDE_REG_CMD`	Command Register.
`FS_DEV_IDE_REG_ALTSTATUS`	Alternate Status Register.

Returned Value

Register value.

Notes/Warnings

None.

9.02.02.06 Write Register on IDE Device: FSDev_IDE_BSP_RegWr()

Write to IDE device register.

Prototype

```
void  FSDev_IDE_BSP_RegWr (FS_QTY      unit_nbr,
                           CPU_INT08U  reg,
                           CPU_INT08U  val);
```

Arguments

`unit_nbr` Unit number of IDE device.

`reg` Register to write :

`FS_DEV_IDE_REG_FR`	Features Register.
`FS_DEV_IDE_REG_SC`	Sector Count Register.
`FS_DEV_IDE_REG_SN`	Sector Number Register.
`FS_DEV_IDE_REG_CYL`	Cylinder High Register.
`FS_DEV_IDE_REG_CYN`	Cylinder Low Register.
`FS_DEV_IDE_REG_DH`	Card/Drive/Head Register.
`FS_DEV_IDE_REG_CMD`	Command Register.
`FS_DEV_IDE_REG_DEVCTRL`	Device Control Register.

`val` Value to write into register.

Returned Value

None.

Notes/Warnings

None.

9.02.02.07 Write Command to IDE Device Registers: FSDev_IDE_BSP_CmdWr()

Write 7-byte command to IDE device registers.

Prototype

```
void  FSDev_IDE_BSP_CmdWr (FS_QTY      unit_nbr,
                           CPU_INT08U  cmd[]);
```

Arguments

`unit_nbr` Unit number of IDE device.

`cmd` Array holding command.

Returned Value

None.

Notes/Warnings

The 7 bytes of the command should be written to the IDE device registers as follows:

```
REG_FR   = Byte 0
REG_SC   = Byte 1
REG_SN   = Byte 2
REG_CYL  = Byte 3
REG_CYN  = Byte 4
REG_DH   = Byte 5
REG_CMD  = Byte 6
```

9.02.02.08 Read Data from IDE Device: FSDev_IDE_BSP_DataRd()

Read data from IDE device.

Prototype

```
void  FSDev_IDE_BSP_DataRd (FS_QTY       unit_nbr,
                            void        *p_dest,
                            CPU_SIZE_T   cnt);
```

Arguments

`unit_nbr`	Unit number of IDE device.
`p_dest`	Pointer to destination memory buffer.
`cnt`	Number of octets of data to read.

Returned Value

None.

Notes/Warnings

None.

9.02.02.09 Write Data to IDE Device: FSDev_IDE_BSP_DataWr()

Write data to IDE device.

Prototype

```
void  FSDev_IDE_BSP_DataWr (FS_QTY       unit_nbr,
                            void        *p_src,
                            CPU_SIZE_T   cnt);
```

Arguments

`unit_nbr`	Unit number of IDE device.
`p_src`	Pointer to source memory buffer.
`cnt`	Number of octets to write.

Returned Value

None.

Notes/Warnings

None.

9.02.02.10 Setup DMA: FSDev_IDE_BSP_DMA_Start()

Setup DMA for command (initialize channel)

Prototype

```
void  FSDev_IDE_BSP_DMA_Start (FS_QTY        unit_nbr,
                               void         *p_data,
                               CPU_SIZE_T    cnt,
                               FS_FLAGS      mode_type,
                               CPU_BOOLEAN   rd);
```

Arguments

`unit_nbr` Unit number of IDE device.

`p_dest` Pointer to memory buffer.

`cnt` Number of octets to transfer.

`mode_type` Transfer mode type:

`FS_DEV_IDE_MODE_TYPE_DMA`	Multiword DMA mode.
`FS_DEV_IDE_MODE_TYPE_UDMA`	Ultra-DMA mode.

`rd` Indicates whether transfer is read or write:

`DEF_YES`	Transfer is read.
`DEF_NO`	Transfer is write.

Returned Value

None.

Notes/Warnings

1. DMA setup occurs before the command is executred (in `FSDev_IDE_BSP_CmdWr()`). Afterwards, data transmission completion must be confirmed (in `FSDev_IDE_BSP_DMA_End()`) before the driver checks the command status.

2. .If the return value of `FSDev_IDE_BSP_GetModesSupported()` does NOT include `FS_DEV_IDE_MODE_TYPE_DMA` or FS_DEV_IDE_MODE_TYPE_UDMA, this function need not be implemented; it will never be called.

9.02.02.11 End DMA Transfer: FSDev_IDE_BSP_DMA_End()

End DMA transfer (and uninitialize channel)

Prototype

```
void  FSDev_IDE_BSP_DMA_End (FS_QTY        unit_nbr,
                             void         *p_data,
                             CPU_SIZE_T    cnt,
                             CPU_BOOLEAN   rd);
```

Arguments

`unit_nbr` Unit number of IDE device.

`p_dest` Pointer to memory buffer.

`cnt` Number of octets to transfer.

`rd` Indicates whether transfer is read or write:

`DEF_YES`	Transfer was read.
`DEF_NO`	Transfer was write.

Returned Value

None.

Notes/Warnings

1. DMA setup occurs before the command is executred (in `FSDev_IDE_BSP_CmdWr()`). Afterwards, data transmission completion must be confirmed (in `FSDev_IDE_BSP_DMA_End()`) before the driver checks the command status.

2. When this function returns, the host controller should be setup to transmit commands in PIO mode.

3. .If the return value of `FSDev_IDE_BSP_GetModesSupported()` does NOT include `FS_DEV_IDE_MODE_TYPE_DMA` or `FS_DEV_IDE_MODE_TYPE_UDMA`, this function need not be implemented; it will never be called.

9.02.02.12 Get IDE Drive Number: FSDev_IDE_BSP_GetDrvNbr()

Get IDE drive number.

Prototype

```
CPU_INT08U  FSDev_IDE_BSP_GetDrvNbr (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

Drive number (0 or 1).

Notes/Warnings

Two IDE devices may be accessed on the same bus by setting the DEV bit of the drive/head register. If the bit should be clear, this function should return 0; if the bit should be set, this function should return 1.

9.02.02.13 Get Supported Transfer Modes: FSDev_IDE_BSP_GetModesSupported()

Get supported transfer modes.

Prototype

```
FS_FLAGS  FSDev_IDE_BSP_GetModesSupported (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

But mapped variable indicating supported transfer mode(s); should be the bitwise OR of one or more of:

`FS_DEV_IDE_MODE_TYPE_PIO`	PIO mode supported.
`FS_DEV_IDE_MODE_TYPE_DMA`	Multiword DMA mode supported.
`FS_DEV_IDE_MODE_TYPE_UDMA`	Ultra-DMA mode supported.

Notes/Warnings

None.

9.02.02.14 Set Transfer Mode: FSDev_IDE_BSP_SetMode()

Set transfer mode timings.

Prototype

```
CPU_INT08U  FSDev_IDE_BSP_SetMode (FS_QTY      unit_nbr,
                                   FS_FLAGS    mode_type,
                                   CPU_INT08U  mode);
```

Arguments

`unit_nbr` Unit number of IDE device.

`mode_type` Transfer mode type:.

`FS_DEV_IDE_MODE_TYPE_PIO`	PIO mode.
`FS_DEV_IDE_MODE_TYPE_DMA`	Multiword DMA mode.
`FS_DEV_IDE_MODE_TYPE_UDMA`	Ultra-DMA mode.

`mode` Transfer mode, between 0 and maximum mode supported for mode type by device (inclusive).

Returned Value

Mode selected; should be between 0 and `mode`, inclusive.

Notes/Warnings

If DMA is supported, two transfer modes will be setup. The first will be a PIO mode; the second will be a Multiword DMA or Ultra-DMA mode. Thereafter, the host controller or bus interface must be capable of both PIO and DMA transfers.

9.02.02.15 Delay 400 ns: FSDev_IDE_BSP_Dly400_ns()

This function delays 400 ns.

Prototype

```
void  FSDev_IDE_BSP_Dly400_ns (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of IDE device.

Returned Value

None.

Notes/Warnings

None.

Chapter

10

Logical Device Driver

The logical device driver is not released yet. It should be released in a soon future.

Chapter

11

MSC Driver

The MSC driver supports USB mass storage class devices (i.e., USB drives, thumb drives) using the µC/USB host stack.

11.01 Files and Directories

The files inside the MSC driver directory are outlined in this section; the generic file-system files, outlined in Section 1.02, are also required.

\Micrium\Software\uC-FS\Dev

This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\MSC

This directory contains the MSC driver files.

fs_dev_msc.* *constitute the MSC device driver.*

\Micrium\Software\uC-USB

This directory contains the code for µC/USB. For more information, please see the µC/USB user manual.

11.02 Using the MSC Driver

To use the MSC driver, two files, in addition to the generic file system files, must be included in the build:

- *fs_dev_msc.c* (located in the directory specified in Section 11.01).
- *fs_dev_msc.h* (located in the directory specified in Section 11.01).

The file *fs_dev_msc.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directory must be on the project include path:

- *\Micrium\Software\uC-FS\Dev\MSC*

Before µC/FS is initialized, the µC/USB host stack must be initialized as shown in Listing 11-1. The file system initialization function (`FS_Init()`) must then be called and the MSC driver, `FSDev_MSC`, restivered (using `FS_DevDrvAdd()`). The USB notification function should add/remove devices when events occur, as shown in Listing 11-1.

ROM/RAM characteristics and performance benchmarks of the MSC driver can be found in Section 8.01.01.

Listing 11-1. Example µC/USB initialization.

```
static  void  App_InitUSB_Host (void)
{
    USBH_ERR  err;


    err = USBH_HostCreate(&App_USB_Host, &USBH_AT91SAM9261_Drv);
    if (err != USBH_ERR_NONE) {
        return;
    }

    err = USBH_HostInit(&App_USB_Host);
    if (err != USBH_ERR_NONE) {
        return;
    }

    USBH_ClassDrvReg(&App_USB_Host, &USBH_MSC_ClassDrv,
                            (USBH_CLASS_NOTIFY_FNCT)App_USB_HostMSC_ClassNotify, (void *)0);
}
```

Listing 11-2. µC/USB MSC notification function.

```
static  void  App_USB_HostMSC_ClassNotify (void        *pclass_dev,
                                           CPU_INT08U   is_conn,
                                           void        *pctx)
{
    USBH_MSC_DEV  *p_msc_dev;
    USBH_ERR       usb_err;
    FS_ERR         fs_err;


    p_msc_dev = (USBH_MSC_DEV *)pclass_dev;
    switch (is_conn) {
        case USBH_CLASS_DEV_STATE_CONNECTED:        /* ----- MASS STORAGE DEVICE CONN'D ----- */
             usb_err = USBH_MSC_RefAdd(p_msc_dev);
             if (usb_err == USBH_ERR_NONE) {
                 FSDev_MSC_DevOpen(p_msc_dev, &fs_err);
             }
             break;


        case USBH_CLASS_DEV_STATE_REMOVED:          /* ----- MASS STORAGE DEVICE REMOVED ---- */
             FSDev_MSC_DevClose(p_msc_dev);
             USBH_MSC_RefRel(p_msc_dev);
             break;


        default:
             break;
    }
}
```

If the file system and USB stack initialization succeed, the file system will produce the trace output as shown in Figure 11-2 (if a sufficiently high trace level is configured) when the a MSC device is connected. See Section D.09 about configuring the trace level.

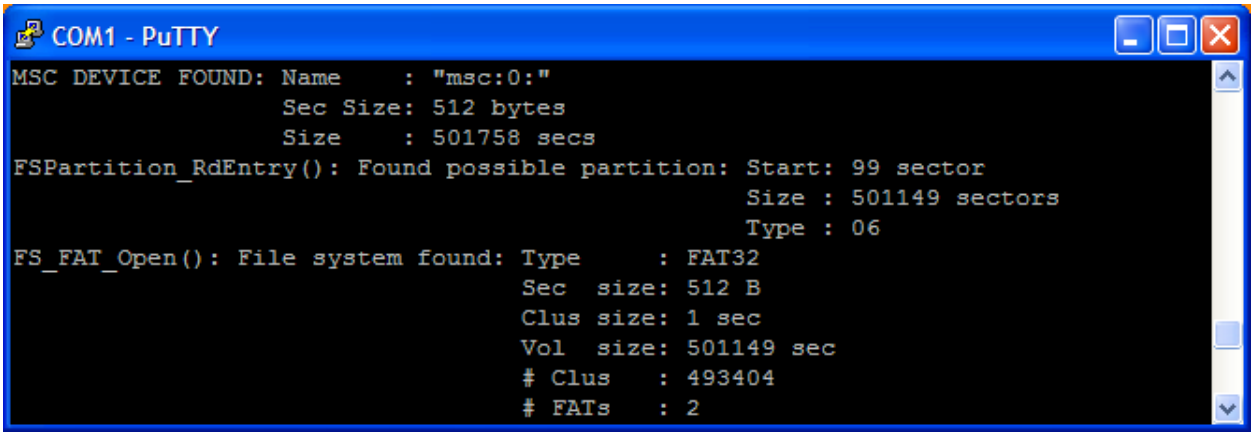


Figure 11-2. MSC Detection Trace Output.

Chapter

12

NAND Flash Driver

The NAND flash driver is not released yet. It should be released in a soon future.

Chapter

13

NOR Flash Driver

NOR flash is a low-capacity on-board storage solution. Traditional parallel NOR flash, located on the external bus of a CPU, offers extremely fast read performance, but comparatively slow writes (typically performed on a word-by-word basis). Often, these store application code in addition to providing a file system; if the application is executed directly from the flash, additional precautions (as covered in Section 13.02) must be taken. The parallel architecture of traditional NOR flash restricts use to a narrow class of CPUs and may consume valuable PCB space. Increasingly, serial NOR flash are a valid alternative, with fast reads speeds and comparable capacities, but demanding less of the CPU and hardware, being accessed by SPI or SPI-like protocols. Table 13-1 briefly compares these two technologies; specific listings of supported devices are located in Section 13.05.

Device Category	Typical Packages	Manufacturers	Description
Parallel NOR Flash	TSOP32, TSOP48, BGA48, TSOP56, BGA56	AMD (Spansion) Intel (Numonyx) SST ST (Numonyx)	Parallel data (8- or 16-bit) and address bus (20+ bits). Most devices have CFI 'query' information and use one of several standard command sets.
Serial NOR Flash	SOIC-8N, SOIC-8W, SOIC-16, WSON, USON	Atmel SST ST (Numonyx)	SPI or multi-bit SPI-like interface. Command sets are generally similar.

Table 13-1. NOR Flash Devices

13.01 Files and Directories

The files inside the RAM disk driver directory are outlined in this section; the generic file-system files, outlined in Section 1.02, are also required.

\Micrium\Software\uC-FS\Dev

This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\NOR

This directory contains the NOR driver files.

fs_dev_nor.* are device driver for NOR flash devices. This file requires a set of BSP functions be defined in a file named ***fs_dev_nor_bsp.c*** to work with a certain hardware setup.

.\BSP\Template\fs_dev_nor_bsp.c is a template BSP for traditional parallel NOR devices. See Section 12.04 for more information.

.\BSP\Template (SPI)\fs_dev_nor_bsp.c is a template BSP for serial (SPI) NOR devices. See Section 12.03 for more information.

.\BSP\Template (SPI GPIO)\fs_dev_nor_bsp.c is a template BSP for serial (SPI) NOR devices using GPIO (bit-banging). See Section 15.03 for more information.

.\PHY

This directory contains physical-level drivers for specific NOR types:

fs_dev_nor_amd_1x08.*	CFI-compatible parallel NOR implementing AMD command set (1 chip, 8-bit data bus)
fs_dev_nor_amd_1x16.*	CFI-compatible parallel NOR implementing AMD command set (1 chip, 16-bit data bus)
fs_dev_nor_intel.*	CFI-compatible parallel NOR implementing Intel command set (1 chip, 16-bit data bus)
fs_dev_nor_sst39.*	SST SST39 Multi-Purpose Flash
fs_dev_nor_stm25.*	ST STM25 serial flash
fs_dev_nor_sst25.*	SST SST25 serial flash

\Micrium\Software\uC-FS\Examples\BSP\Dev\NOR

Each subdirectory contains an example BSP for a particular platform. These are named according to the following rubric:

<Chip Manufacturer>\<Board or CPU>\fs_dev_nor_bsp.c

13.02 Driver & Device Characteristics

NOR devices, no matter what attachment interface (serial or parallel), share certain characteristics. The medium is always organized into units (called blocks) which are erased at the same time; when erased, all bits are 1. Only an erase operation can change a bit from a 0 to a 1, but any bit can be individually programmed from a 1 to a 0. The µC/FS driver requires that any 2-byte word can be individually accessed (read or programmed).

The driver RAM requirement depends on flash parameters such as block size and run-time configurations such as sector size. For a particular instance, a general formula can give an approximate:

```
if (secs_per_blk < 255) {
    temp1 = ceil(blk_cnt_used / 8) + (blk_cnt_used * 1);
} else {
    temp1 = ceil(blk_cnt_used / 8) + (blk_cnt_used * 2);
}

if (sec_cnt < 65535) {
    temp2 = sec_cnt * 2;
} else {
    temp2 = sec_cnt * 4;
}

temp3 = sec_size;

TOTAL = temp1 + temp2 + temp3;
```

where

secs_per_blk	The number of sectors per block.
blk_cnt_used	The number of blocks on the flash which will be used for the file system.
sec_cnt	The total number of sectors on the device.
sec_size	The sector size configured for the device, in octets.

secs_per_blk and sec_cnt can be calculated from more basic parameters:

```
secs_per_blk = floor(blk_size / sec_size);

sec_cnt      = secs_per_blk * blk_cnt_used;
```

where

blk_size	The size of a block on the device, in octets

Take as an example a 16-Mb NOR that is entirely dedicated to file system usage, with a 64-KB block size, configured with a 512-B sector. The following parameters describe the format:

```
blk_cnt_used = 32;
blk_size     = 65536;
sec_size     = 512;
secs_per_blk = 65536 / 512 = 128;
sec_cnt      = 128 * 32    = 4096;
```

and the RAM usage is approximately

```
temp1 = (32 / 8) + (32 * 2) = 68;
temp2 = 4096 * 2 = 8192;
temp3 = 512;

TOTAL = 68 + 8192 + 512 = 8772;
```

In this example, as in most situations, increasing the sector size will decrease the RAM usage. If the sector size were 1024-B, only 5188-B would have been needed, but a moderate performance penalty would be paid.

13.03 Using a Parallel NOR Device

To use the NOR driver, five files, in addition to the generic file system files, must be included in the build:

- *fs_dev_nor.c* (located in the directory specified in Section 13.01).
- *fs_dev_nor.h* (located in the directory specified in Section 13.01).
- *fs_dev_nor_bsp.c* (located in the user application or BSP).
- A physical-layer driver, typically one provided in *\Micrium\Software\uC-FS\Dev\NOR\PHY*

The file *fs_dev_nor.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directories must be on the project include path:

- *\Micrium\Software\uC-FS\Dev\NOR*
- *\Micrium\Software\uC-FS\Dev\NOR\PHY*

A single NOR volume is opened as shown in Listing 13-1. The file system initialization (`FS_Init()`) function must have previously been called.

ROM characteristics and performance benchmarks of the NOR driver can be found in Section 8.01.01. The NOR driver also provides interface functions to perform low-level operations (see Section A.13).

Listing 13-1. Opening a NOR device volume.

```
CPU_BOOLEAN  App_FS_AddNOR (void)
{
    FS_DEV_NOR_CFG  nor_cfg;
    FS_ERR          err;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_Nor,           /* (1)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) && (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        return (DEF_FAIL);
    }

                                                     /* (2)    */
    nor_cfg.AddrBase         =  APP_CFG_FS_NOR_ADDR_BASE;
    nor_cfg.RegionNbr        =  APP_CFG_FS_NOR_REGION_NBR;

    nor_cfg.AddrStart        =  APP_CFG_FS_NOR_ADDR_START;
    nor_cfg.DevSize          =  APP_CFG_FS_NOR_DEV_SIZE;
    nor_cfg.SecSize          =  APP_CFG_FS_NOR_SEC_SIZE;
    nor_cfg.PctRsvd          =  APP_CFG_FS_NOR_PCT_RSVD;
    nor_cfg.PctRsvdSecActive =  APP_CFG_FS_NOR_PCT_RSVD_SEC_ACTIVE;
    nor_cfg.EraseCntDiffTh   =  APP_CFG_FS_NOR_ERASE_CNT_DIFF_TH;

    nor_cfg.PhyPtr           = (FS_DEV_NOR_PHY_API *)APP_CFG_FS_NOR_PHY_PTR;

    nor_cfg.BusWidth         =  APP_CFG_FS_NOR_BUS_WIDTH;
    nor_cfg.BusWidthMax      =  APP_CFG_FS_NOR_BUS_WIDTH_MAX;
    nor_cfg.PhyDevCnt        =  APP_CFG_FS_NOR_PHY_DEV_CNT;
    nor_cfg.MaxClkFreq       =  APP_CFG_FS_NOR_MAX_CLK_FREQ;

                                                     /* (3)    */
    FSDev_Open((CPU_CHAR *)"nor:0:",                 /*    (a) */
               (void     *)&nor_cfg,                 /*    (b) */
               (FS_ERR   *)&err);

    switch (err) {
        case FS_ERR_NONE:
             APP_TRACE_DBG((" ...opened device.\r\n"));
             break;
```

```
        case FS_ERR_DEV_INVALID_LOW_FMT:            /* Low fmt invalid.    */
             APP_TRACE_DBG("    ...opened device (not low-level formatted).\r\n");
             FSDev_NOR_LowFmt("nor:0:", &err);      /* (4)     */
             if (err != FS_ERR_NONE) {
                APP_TRACE_DBG("    ...low-level format failed.\r\n");
                return (DEF_FAIL);
             }
             break;

        default:                                    /* Device error.       */
             APP_TRACE_DBG("    ...opening device failed w/err = %d.\r\n\r\n", err);
             return (DEF_FAIL);
    }
                                                    /* (5)     */
    FSVol_Open((CPU_CHAR         *)"nor:0:",        /*    (a)  */
               (CPU_CHAR         *)"nor:0:",        /*    (b)  */
               (FS_PARTITION_NBR  ) 0,              /*    (c)  */
               (FS_ERR           *)&err);

    switch (err) {
        case FS_ERR_NONE:
             APP_TRACE_DBG("    ...opened volume (mounted).\r\n");
             break;

        case FS_ERR_PARTITION_NOT_FOUND:            /* Volume error.       */
             APP_TRACE_DBG("    ...opened device (not formatted).\r\n");
             FSVol_Fmt("nor:0:", (void *)0, &err);  /* (6)     */
             if (err != FS_ERR_NONE) {
                APP_TRACE_DBG("    ...format failed.\r\n");
                return (DEF_FAIL);
             }
             break;

        default:                                      /* Device error.       */
             APP_TRACE_DBG("    ...opening volume failed w/err = %d.\r\n\r\n", err);
             return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

L13-1(1) Register the NOR device driver `FSDev_NOR`.

L13-1(2) The NOR device configuration should be assigned. For more information about these parameters, see Section C.10.

L13-1(3) `FSDev_Open()` opens/initializes a file system device. The parameters are the device name (3a) and a pointer to a device driver-specific configuration structure (3b). The device name (3a) s composed of a device driver name (“nor”), a single colon, an ASCII-formatted integer (the unit number) and another colon.

L13-1(4) `FSDev_NOR_LowFmt()` low-level formats a NOR. If the NOR has never been used with µC/FS, it must be low-level formatted before being used. Low-level formatting will associate logical sectors with physical areas of the device.

L13-1(5) `FSVol_Open()` opens/mounts a volume. The parameters are the volume name (5a), the device name (5b) and the partition that will be opened (5c). There is no restriction on the volume name (5a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partition, then the partition number (5c) should be zero.

L13-1(6) `FSVol_Fmt()` formats a file system device. If the NOR has just been low-level format, it will have no file system on it after it is opened (it will be unformatted) and must be formatted before files can be created or accessed.

If the NOR initialization succeeds, the file system will produce the trace output as shown in Figure 13-3 (if a sufficiently high trace level is configured). See Section D.09 about configuring the trace level.

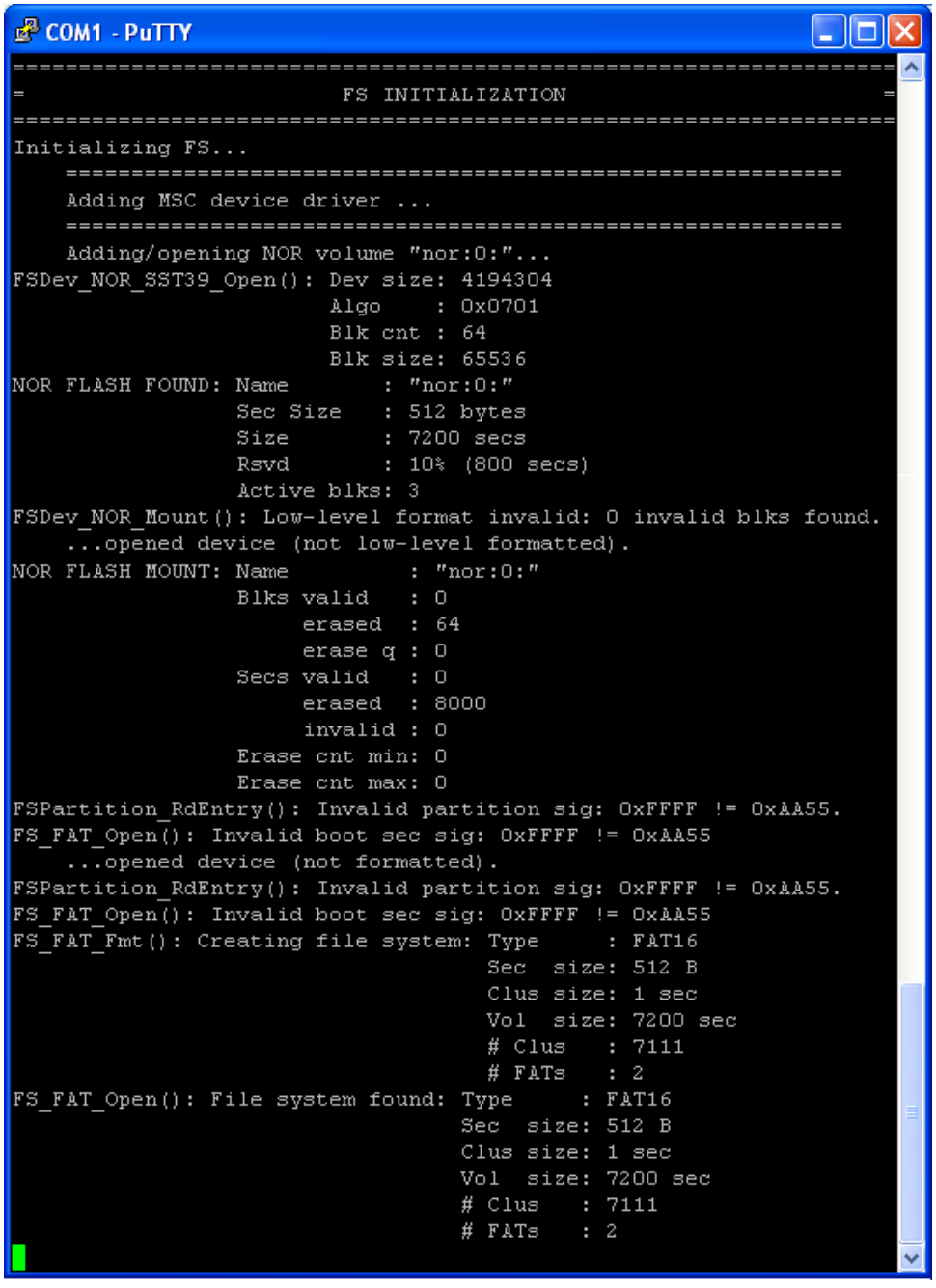


Figure 13-3. NOR detection trace output.

13.03.01 Driver Architecture

When used with a parallel NOR device, the NOR driver is three layered, as depicted in the figure below. The generic NOR driver, as always, provides sector abstraction and performs wear-leveling (to make certain all blocks are used equally). Below this, the physical-layer driver implements a particular command set to read and program the flash and erase blocks. Lastly, a BSP implements function to initialize and unitialize the bus interface. Device commands are executed by direct access to the NOR, at locations appropriately offset from the configured base address.

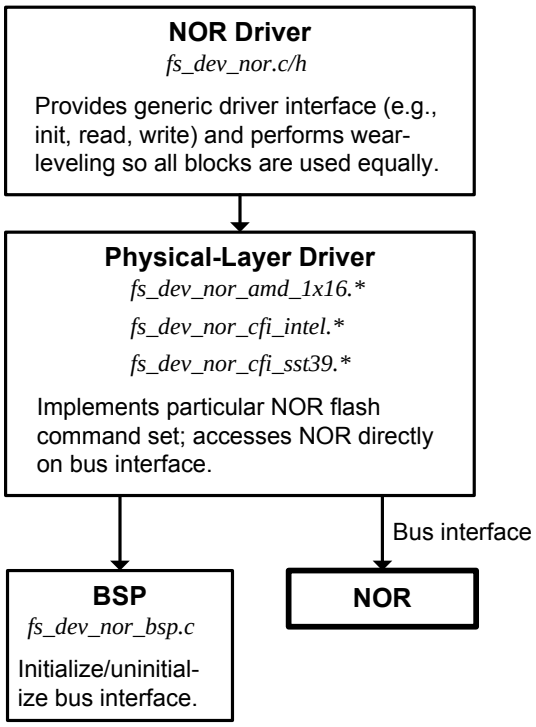


Figure 13-2. NOR driver architecture (parallel NOR flash).

13.03.02 Hardware

Parallel NOR devices typically connect to a host MCU/MPU via an external bus interface (EBI), with an 8- or 16-bit data lines and 20 or more address lines (depending on the device size). Many silicon vendors offer parallel NOR product lines; most devices currently marketed are conformant to the Common Flash Interface (CFI). A set of query information allows the µC/FS NOR driver physical-layer drivers to interface with almost any NOR flash without configuration or modification. The standard query information provides the following details:

- **Command set**. Three different command sets are common: Intel, AMD and SST. All three are supported.
- **Geometry**. A device is composed of one or more regions of identically-sized erase blocks. Uniform devices contain only one region. Boot-block devices often have one or two regions of small blocks for code storage at the top or bottom of the device. All of these are supported by the NOR driver.

Offset	Length (Bytes)	Contents
0x10	1	Query string “Q”
0x11	1	Query string “R”
0x12	1	Query string “Y”
0x13	2	Command set
0x27	1	Device size, in bytes = 2^n
0x2A	2	Maximum number of bytes in multi-byte write = 2^N
0x2C	1	Number of erase block regions = *m*
0x2D	2	Region 1: Number of erase blocks = *x* + 1
0x2F	2	Region 1: Size of each erase block = *y* * 256 (bytes)
0x31	2	Region 2: Number of erase blocks = *x* + 1
0x33	2	Region 2: Size of each erase block = *y* * 256 (bytes)
		.
		.
		.
0x2D + (m-1) * 4	2	Region *m*: Number of erase blocks = *x* + 1
0x2F + (m-1) * 4	2	Region *m*: Size of each erase block = *y* * 256 (bytes)

Table 13-2. CFI query information

Table 13-2 gives the format of CFI query information. The first three bytes should constitute the marker string “QRY”, by which the retrieval of correct parameters is verified. A two-byte command set identifier follows; this must match the identifier for the command set supported by the physical-layer driver. Beyond is the geometry information: the device size, the number of erase block regions, and the size and number of blocks in each region. For most flash, these regions are contiguous and sequential, the first at the beginning of the device, the second just after. Since this is not always true (see Section 13.05.03 for an example), the manufacturer’s information should always be checked and, for atypical devices, the physical-layer driver copied to the application directory and modified.

Command Set Identifier	Description
0x0001	Intel
0x0002	AMD/Spansion
0x0003	Intel
0x0102	SST

Table 13-3. Common command sets

13.03.03 NOR BSP Overview

A BSP is required so that a physical-layer driver for a parallel flash will work on a particular system. The functions shown in the table below must be implemented.

Function	Description
`FSDev_NOR_BSP_Open()`	Open (initialize) bus for NOR.
`FSDev_NOR_BSP_Close()`	Close (uninitialize) bus for NOR.
`FSDev_NOR_BSP_Rd_XX()`	Read from bus interface.
`FSDev_NOR_BSP_RdWord_XX()`	Read word from bus interface.
`FSDev_NOR_BSP_WrWord_XX()`	Write word to bus interface
`FSDev_NOR_BSP_WaitWhileBusy()`	Wait while NOR is busy.

Table 13-4. NOR BSP functions

The `Open()`/`Close()` functions are called upon open/close; these calls are always matched.

The remaining functions (`Rd_XX()`, `RdWord_XX()`, `WrWord_XX()`) read data from or write data to the NOR. If a single parallel NOR device will be accessed, these function may be defined as macros to speed up bus accesses.

13.03.03.01 Open (Initialize) Bus for NOR: FSDev_NOR_BSP_Open( )

Open (initialize) bus for NOR.

Prototype

```
CPU_BOOLEAN  FSDev_NOR_BSP_Open (FS_QTY      unit_nbr,
                                 CPU_ADDR    addr_base,
                                 CPU_INT08U  bus_width,
                                 CPU_INT08U  phy_dev_cnt);
```

Arguments

`unit_nbr` Unit number of NOR.

`addr_base` Base address of NOR.

`bus_width` Bus width, in bits.

`phy_dev_cnt` Number of devices interleaved.

Returned Value

`DEF_OK`, if interface was opened.
`DEF_FAIL`, otherwise.

Notes/Warnings

This function will be called every time the device is opened.

13.03.03.02 Close (Uninitialize) Bus for NOR: FSDev_NOR_BSP_Close()

Close (uninitialize) bus for NOR.

Prototype

```
void   FSDev_NOR_BSP_Close (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of NOR.

Returned Value

None.

Notes/Warnings

This function will be called every time the device is closed.

13.03.03.03 Read from Bus Interface: FSDev_NOR_BSP_Rd_XX()

Read from bus interface.

Prototype

```
void  FSDev_NOR_BSP_Rd_08 (FS_QTY       unit_nbr,
                           void        *p_dest,
                           CPU_ADDR     addr_src,
                           CPU_INT32U   cnt);
```

```
void  FSDev_NOR_BSP_Rd_16 (FS_QTY       unit_nbr,
                           void        *p_dest,
                           CPU_ADDR     addr_src,
                           CPU_INT32U   cnt);
```

Arguments

`unit_nbr`	Unit number of NOR.
`p_dest`	Pointer to destination memory buffer.
`addr_src`	Source address.
`cnt`	Number of words to read.

Returned Value

None.

Notes/Warnings

Data should be read from the bus in words sized to the data bus; for any unit, only the function with that access width will be called.

13.03.03.04 Read Word from Bus Interface: FSDev_NOR_BSP_RdWord_XX()

Read word from bus interface.

Prototype

```
CPU_INT08U  FSDev_NOR_BSP_RdWord_08 (FS_QTY    unit_nbr,
                                     CPU_ADDR  addr_src);

CPU_INT16U  FSDev_NOR_BSP_RdWord_16 (FS_QTY    unit_nbr,
                                     CPU_ADDR  addr_src);
```

Arguments

`unit_nbr` Unit number of NOR.

`addr_src` Source address.

Returned Value

Word read.

Notes/Warnings

Data should be read from the bus in words sized to the data bus; for any unit, only the function with that access width will be called.

13.03.03.05 Write Word to Bus Interface: FSDev_NOR_BSP_WrWord_XX()

Write word to bus interface.

Prototype

```
void  FSDev_NOR_BSP_WrWord_08 (FS_QTY      unit_nbr,
                               CPU_ADDR    addr_dest,
                               CPU_INT08U  datum);

void  FSDev_NOR_BSP_WrWord_16 (FS_QTY      unit_nbr,
                               CPU_ADDR    addr_dest,
                               CPU_INT08U  datum);
```

Arguments

`unit_nbr` Unit number of NOR.

`addr_dest` Source address.

`datum` Word to write.

Returned Value

None.

Notes/Warnings

Data should be written from the bus in words sized to the data bus; for any unit, only the function with that access width will be called.

13.03.03.06 Wait while NOR is Busy: FSDev_NOR_BSP_WaitWhileBusy()

Wait while NOR is busy.

Prototype

```
void  FSDev_NOR_BSP_WaitWhileBusy(FS_QTY                 unit_nbr,
                                  FS_DEV_NOR_PHY_DATA  *p_phy_data,
                                  CPU_BOOLEAN         (*poll_fnct)
                                                      (FS_DEV_NOR_PHY_DATA *),
                                  CPU_INT32U            to_us);
```

Arguments

`unit_nbr` Unit number of NOR.

`p_phy_data` Pointer to NOR phy data.

`poll_fnct` Pointer to function to poll, if there is no hardware ready/busy signal.

`to_us` Timeout, in microseconds.

Returned Value

`DEF_OK`, if NOR became ready.
`DEF_FAIL`, otherwise.

Notes/Warnings

None.

13.04 Using a Serial NOR Device

When used with a serial NOR device, the NOR driver is three layered, as depicted in the figure below. The generic NOR driver, as always, provides sector abstraction and performs wear-leveling (to make certain all blocks are used equally). Below this, the physical-layer driver implements a particular command set to read and program the flash and erase blocks. Lastly, a BSP implements function to communicate with the device over SPI. Device commands are executed though this BSP.

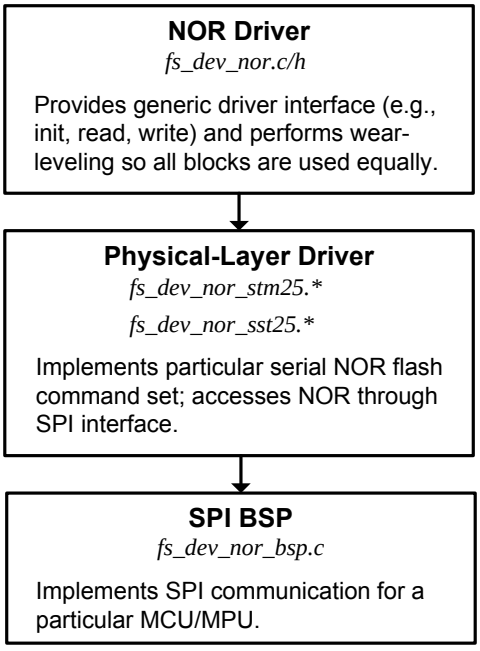


Figure 13-4. NOR driver architecture (serial NOR flash).

13.04.01 Hardware

Serial NOR devices typically connect to a host MCU/MPU via an SPI or SPI-like bus. Eight-pin devices, with the functions listed in Table 13-4, or similar, are common, and are often employed with the HOLD and WP pins held high (logic low, or inactive), as shown in Figure 13-2. As with any SPI device, four signals are used to communicate with the host (CS, SI, SCK and SO).

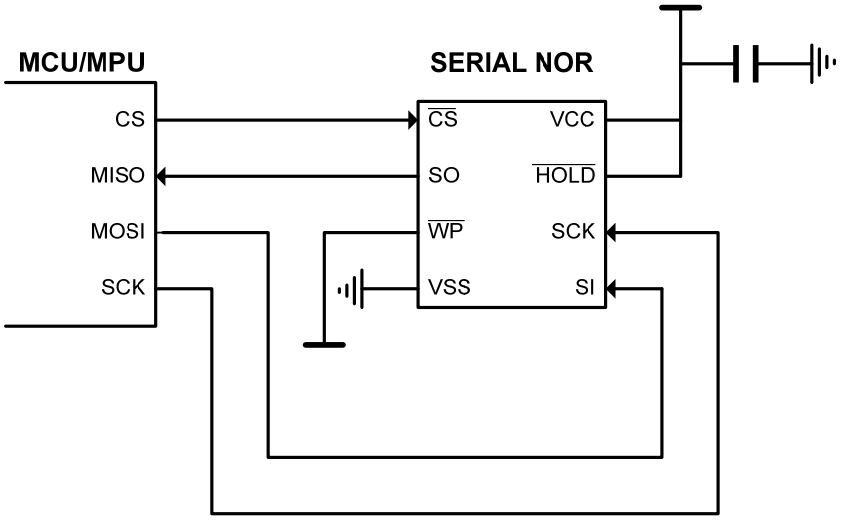


Figure 13-2. Typical serial NOR connections.

13.04.02 NOR SPI BSP Overview

An NOR BSP is required so that a physical-layer driver for a serial flash will work on a particular system. For more information about these functions, see Section 8.03.

Function	Description
`FSDev_NOR_BSP_SPI_Open()`	Open (initialize) SPI.
`FSDev_NOR_BSP_SPI_Close()`	Close (uninitialize) SPI.
`FSDev_NOR_BSP_SPI_Lock()`	Acquire SPI lock.
`FSDev_NOR_BSP_SPI_Unlock()`	Release SPI lock.
`FSDev_NOR_BSP_SPI_Rd()`	Read from SPI.
`FSDev_NOR_BSP_SPI_Wr()`	Write to SPI.
`FSDev_NOR_BSP_SPI_ChipSelEn()`	Enable chip select.
`FSDev_NOR_BSP_SPI_ChipSelDis()`	Disable chip select.
`FSDev_NOR_BSP_SPI_SetClkFreq()`	Set SPI clock frequency.

Table 13-5. NOR SPI BSP Functions

13.05 Physical-Layer Drivers

The physical-layer drivers distributed with the NOR driver (see the table below) support a wide variety of parallel and serial flash devices from major vendors. Whenever possible, advanced programming algorithms (such as the common buffered programming commands) are used to optimize performance. Within the diversity of NOR flash, some may be found which implement the basic command set, but not the advanced features; for these, a released physical-layer may need to be modified. In all cases, the manufacturer's reference should be compared to the driver description below.

Driver API	Files	Description
`FSDev_NOR_AMD_1x08`	*fs_dev_nor_amd_1x08.**	Supports CFI-compatible devices with 8-bit data bus implementing AMD command set.
`FSDev_NOR_AMD_1x16`	*fs_dev_nor_amd_1x16.**	Supports CFI-compatible devices i with 16-bit data bus mplementing AMD command set.
`FSDev_NOR_Intel_1x16`	*fs_dev_nor_intel.**	Supports CFI-compatible devices i with 16-bit data bus mplementing Intel command set.
`FSDev_NOR_SST39`	*fs_dev_nor_sst39.**	Supports various SST SST39 devices with 16-bit data bus.
`FSDev_NOR_STM29_1x08`	*fs_dev_nor_stm29_1x08.**	Supports various ST M29 devices with 8-bit data bus.
`FSDev_NOR_STM29_1x16`	*fs_dev_nor_stm29_1x16.**	Supports various ST M29 devices with 16-bit data bus.
`FSDev_NOR_STM25`	*fs_dev_nor_stm25.**	Supports various ST M25 serial devices.
`FSDev_NOR_SST25`	*fs_dev_nor_sst25.**	Supports various SST SST25 serial devices.

Table 13-6. Physical-layer drivers

13.05.01 `FSDev_NOR_AMD_1x08`, `FSDev_NOR_AMD_1x16`

`FSDev_NOR_AMD_1x08` and `FSDev_NOR_AMD_1x16` support CFI NOR flash implementing AMD command set, including

- Most AMD and Spansion devices
- Most ST/Numonyx devices
- Others

The fast programming command "write to buffer and program", supported by many flash implementing the AMD command set, is used in this driver if the "Maximum number of bytes in a multi-byte write" (in the CFI device geometry definition) is non-zero.

Some flash implementing AMD command set have non-zero multi-byte write size but do not support the "write to buffer & program" command. Often these devices will support alternate fast programming methods. This driver MUST be modified for those devices, to ignore the multi-byte write size in the CFI information. Define `NOR_NO_BUF_PGM` to force this mode of operation.

13.05.02 `FSDev_NOR_Intel_1x16`

`FSDev_NOR_Intel_1x16` supports CFI NOR flash implementing Intel command set, including

- Most Intel/Numonyx devices
- Some ST/Numonyx M28 device
- Others

13.05.03 FSDev_NOR_SST39

FSDev_NOR_SST39 supports SST's SST39 Multi-Purpose Flash memories, as described in various datasheets at SST (http://www.sst.com). SST39 devices use a modified form of the AMD command set. A more significant deviation is in the CFI device geometry information, which describes two different views of the memory organization—division in to small sectors and division into large blocks—rather than contiguous, separate regions. The driver always uses the block organization.

13.05.04 FSDev_NOR_STM25

FSDev_NOR_STM25 supports Numonyx/ST's M25 & M45 serial flash memories, as described in various datasheets at Numonyx (http://www.numonyx.com). This driver has been tested with or should work with the devices in the table below.

The M25P-series devices are programmed on a page (256-byte) basis and erased on a sector (32- or 64-KB) basis. The M25PE-series devices are also programmed on a page (256-byte) basis, but are erased on a page, subsector (4-KB) or sector (64-KB) basis.

Manufacturer	Device	Capacity	Block Size	Block Count
ST	M25P10	1 Mb	64-KB	2
ST	M25P20	2 Mb	64-KB	4
ST	M25P40	4 Mb	64-KB	8
ST	M25P80	8 Mb	64-KB	16
ST	M25P16	16 Mb	64-KB	32
ST	M25P32	32 Mb	64-KB	64
ST	M25P64	64 Mb	64-KB	128
ST	M25P128	128 Mb	64-KB	256
ST	M25PE10	1 Mb	64-KB	2
ST	M25PE20	2 Mb	64-KB	4
ST	M25PE40	4 Mb	64-KB	8
ST	M25PE80	8 Mb	64-KB	16
ST	M25PE16	16 Mb	64-KB	32

Table 13-7. Supported M25 serial flash

13.05.05 FSDev_NOR_SST25

FSDev_NOR_SST25 supports SST's SST25 serial flash memories, as described in various datasheets at Numonyx (http://www.numonyx.com). This driver has been tested with or should work with the devices in the table below.

The M25P-series devices are programmed on a word (2-byte) basis and erased on a sector (4-KB) or block (32-KB) basis. The revision A devices and revision B devices differ slightly. Both have an Auto-Address Increment (AAI) programming mode. In revision A devices, the programming is performed byte-by-byte; in revision B devices, word-by-word. Revision B devices can also be erased on a 64-KB block basis and support a command to read a JEDEC-compatible ID.

Manufacturer	Device	Capacity	Block Size	Block Count
SST	SST25VF010B	1 Mb	4-KB	32
SST	SST25VF020B	2 Mb	4-KB	64
SST	SST25VF040B	4 Mb	4-KB	128
SST	SST25VF080B	8 Mb	32-KB	32
SST	SST25VF016B	16 Mb	32-KB	64
SST	SST25VF032B	32 Mb	32-KB	128

Table 13-8. Supported SST25 serial flash

Chapter

14

RAM Disk Driver

The simplest device driver is the RAM disk driver, which uses a block of memory (internal or external) as a storage medium.

14.01 Files and Directories

The files inside the RAM disk driver directory are outlined in this section; the generic file-system files, outlined in Section 1.02, are also required.

\Micrium\Software\uC-FS\Dev

This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\RAMDisk

This directory contains the RAM disk driver files.

fs_dev_ramdisk.* constitute the RAM disk device driver.

14.02 Using the RAM Disk Driver

To use the RAM disk driver, two files, in addition to the generic FS files, must be included in the build:

- *fs_dev_ramdisk.c* (located in the directory specified in Section 14.01).
- *fs_dev_ramdisk.h* (located in the directory specified in Section 14.01).

The file *fs_dev_ramdisk.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directory must be on the project include path:

- *\Micrium\Software\uC-FS\Dev\RAMDisk*

A single RAM disk is opened as shown in Listing 14-1. The file system initialization (`FS_Init()`) function must have previously been called.

ROM/RAM characteristics and performance benchmarks of the RAM disk driver can be found in Section 8.01.01. For more information about the `FS_DEV_RAM_CFG` structure, see Section C.10.

Listing 14-1. Opening a RAM disk volume.

```
#define  APP_CFG_FS_RAM_SEC_SIZE            512      /* (1)    */
#define  APP_CFG_FS_RAM_NBR_SECS     (48 * 1024)

static  CPU_INT32U  App_FS_RAM_Disk[APP_CFG_FS_RAM_SEC_SIZE * APP_CFG_FS_RAM_NBR_SECS / 4];


CPU_BOOLEAN  App_FS_AddRAM (void)
{
    FS_ERR          err;
    FS_DEV_RAM_CFG  cfg;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_RAM,           /* (2)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) && (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        return (DEF_FAIL);
    }

    ram_cfg.SecSize =  APP_CFG_FS_RAM_SEC_SIZE;      /* (3)    */
    ram_cfg.Size    =  APP_CFG_FS_RAM_NBR_SECS;
    ram_cfg.DiskPtr = (void *)&App_FS_RAM_Disk[0]

                                                     /* (4)    */
    FSDev_Open((CPU_CHAR *)"ram:0:",                 /*    (a) */
               (void     *)&ram_cfg,                 /*    (b) */
               (FS_ERR   *)&err);
    if (err != FS_ERR_NONE) {
        return (DEF_FAIL);
    }

                                                    /* (5)    */
    FSVol_Open((CPU_CHAR         *)"ram:0:",        /*    (a) */
               (CPU_CHAR         *)"ram:0:",        /*    (b) */
               (FS_PARTITION_NBR  ) 0,              /*    (c) */
               (FS_ERR           *)&err);

    switch (err) {
        case FS_ERR_NONE:
             APP_TRACE_DBG((    ...opened volume (mounted).\r\n"));
             break;
```

```
        case FS_ERR_PARTITION_NOT_FOUND:            /* Volume error.      */
             APP_TRACE_DBG(("    ...opened device (not formatted).\r\n"));

             FSVol_Fmt("ram:0:", (void *)0, &err);  /* (6)    */
             if (err != FS_ERR_NONE) {
                APP_TRACE_DBG(("    ...format failed.\r\n"));
                return (DEF_FAIL);
             }
             break;

        default:                                    /* Device error.      */
             APP_TRACE_DBG(("    ...opening volume failed w/err = %d.\r\n\r\n", err));
             return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

L14-1(1) The sector size and number of sectors in the RAM disk must be defined. The sector size should be 512, 1024, 2048 or 4096; the number of sectors will be determined by your application requirements. This defines a 24-MB RAM disk (49152 512-B sectors). On most CPUs, it is beneficial to 32-bit align the RAM disk, since this will speed up access.

L14-1(2) Register the RAM disk driver `FSDev_RAM`.

L14-1(3) The RAM disk parameters—sector size, size (in sectors) and pointer to the disk—should be assigned to a `FS_DEV_RAM_CFG` structure.

L14-1(4) `FSDev_Open()` opens/initializes a file system device. The parameters are the device name (3a) and a pointer to a device driver-specific configuration structure (3b). The device name (3a) s composed of a device driver name (“ram”), a single colon, an ASCII-formatted integer (the unit number) and another colon.

L14-1(5) `FSVol_Open()` opens/mounts a volume. The parameters are the volume name (5a), the device name (5b) and the partition that will be opened (5c). There is no restriction on the volume name (5a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partition, then the partition number (5c) should be zero.

L14-1(6) `FSVol_Fmt()` formats a file system volume. If the RAM disk is in volatile RAM, it have no file system on it after it is opened (it will be unformatted) and must be formatted before a volume on it is opened.

If the RAM disk initialization succeeds, the file system will produce the trace output as shown in Figure 14-2 (if a sufficiently high trace level is configured). See Section D.09 about configuring the trace level.

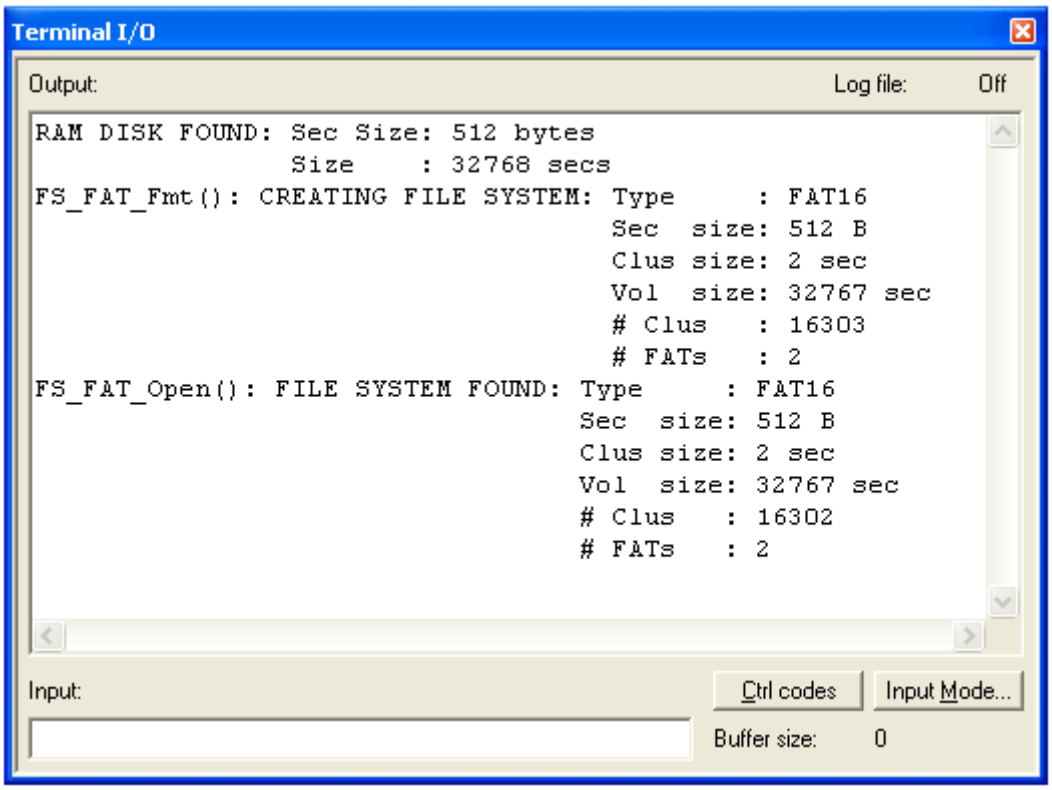


Figure 14-2. RAM Disk Initialization Trace Output.

Chapter

15

SD/MMC Drivers

SD (Secure Digital) cards and MMCs (MultiMedia Cards) are portable, low-cost media often used for storage in consumer devices. Six variants, as shown in Table 15-1, are widely available to electronic retail outlets, all supported by SD/MMC driver. The MMCplus and SD or SDHC are offered in compatible large card formats. Adapters are offered for the remaining devices so that these can fit in standard SD/MMC card slots.

Two further products incorporating SD/MMC technology are emerging. First, some cards now integrate both USB and SD/MMC connectivity, for increased ease-of-access in both PCs and embedded devices. The second are embedded MMC (trademarked eMMC), fixed flash-based media addressed like MMC cards.

Card		Size	Pin Count	Description
MMCPlus		32 x 24 x 1.4 mm	13	Most current MMC cards can operate with 1, 4 or 8 data lines, though legacy media were limited to a single data line. The maximum clock frequency is 20 MHz, providing for maximum theoretical transfer speeds of 20 MB/s, 80 MB/s and 160 MB/s for the three possible bus widths.
MMCmobile		18 x 24 x 1.4 mm	13	
MMCmicro		14 x 12 x 1.1 mm	13	

Table 15-1. SD/MMC Devices

Card		Size	Pin Count	Description
SD or **SDHC**		32 x 24 x 1.4 mm	9	SD cards can operate in cardmode with 1 or 4 data lines or in SPI mode. The maximum clock frequency is 25 MHz, providing for maximum theoretical transfer speeds of 25 MHz and 50 MHz for the two possible bus widths.
SDmini		21.5 x 20 x 1.4 mm	11	
SDmicro		15 x 11 x 1.0 mm	8	

Table 15-1. SD/MMC Devices (continued)

SD/MMC cards can be used in two modes: **card mode** (also referred to as MMC mode and SD mode) and **SPI mode**. The former offers up to 8 data lines (depending on the type of card); the latter, only one data line, but the accessibility of a communication bus common on many MCUs/MPUs. Because these modes involve different command protocols, they require different drivers.

15.01 Files and Directories

The files inside the SD/MMC driver directory is outlined in this section; the generic file-system files, outlined in Section 1.02, are also required.

\Micrium\Software\uC-FS\Dev

This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\SD

This directory contains the SD/MMC driver files.

fs_dev_sd.* contain functions and definitions required for both SPI and card modes.

\Micrium\Software\uC-FS\Dev\SD\Card

This directory contains the SD/MMC driver files for card mode.

fs_dev_sd_card.* are device driver for SD/MMC cards using card mode. This file requires a set of BSP functions be defined in a file named ***fs_dev_sd_card_bsp.c*** to work with a certain hardware setup.

.\BSP\Template\fs_dev_sd_card_bsp.c is a template BSP. See Section 15.04 for more information.

\Micrium\Software\uC-FS\Dev\SD\SPI

This directory contains the SD/MMC driver files for SPI mode.

fs_dev_sd_spi.* are device driver for SD/MMC cards using SPI mode. This file requires a set of BSP functions be defined in a file named ***fs_dev_sd_spi_bsp.c*** to work with a certain hardware setup.

.\BSP\Template\fs_dev_sd_spi_bsp.c is a template BSP. See Section 15.03 for more information.

.\BSP\Template (GPIO)\fs_dev_sd_spi_bsp.c is a template GPIO (bit-banging) BSP. See Section 15.03 for more information.

\Micrium\Software\uC-FS\Examples\BSP\Dev\SD\Card

Each subdirectory contains an example BSP for a particular platform. These are named according to the following rubric:

<Chip Manufacturer>\<Board or CPU>\fs_dev_sd_card_bsp.c

\Micrium\Software\uC-FS\Examples\BSP\Dev\SD\SPI

Each subdirectory contains an example BSP for a particular platform. These are named according to the following rubric:

<Chip Manufacturer>\<Board or CPU>\fs_dev_sd_spi_bsp.c

15.02 Using the SD/MMC CardMode Driver

To use the SD/MMC cardmode driver, five files, in addition to the generic file system files, must be included in the build:

- *fs_dev_sd.c* (located in the directory specified in Section 15.01).
- *fs_dev_sd.h* (located in the directory specified in Section 15.01).
- *fs_dev_sd_card.c* (located in the directory specified in Section 15.01).
- *fs_dev_sd_card.h* (located in the directory specified in Section 15.01).
- *fs_dev_sd_card_bsp.c* (located in the user application or BSP).

The file *fs_dev_sd_card.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directories must be on the project include path:

- *\Micrium\Software\uC-FS\Dev\SD*
- *\Micrium\Software\uC-FS\Dev\SD\Card*

A single SD/MMC volume is opened as shown in Listing 15-1. The file system initialization (`FS_Init()`) function must have previously been called.

ROM/RAM characteristics and performance benchmarks of the SD/MMC driver can be found in Section 8.01.01. The SD/MMC driver also provides interface functions to get low-level card information and read the Card ID and Card-Specific Data registers (see Section A.11).

Listing 15-1. Opening a SD/MMC device volume.

```
CPU_BOOLEAN  App_FS_AddSD_Card (void)
{
    FS_ERR       err;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_SD_Card,        /* (1)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) && (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        return (DEF_FAIL);
    }
                                                      /* (2)    */
    FSDev_Open((CPU_CHAR *)"sdcard:0:",               /*    (a) */
               (void     *) 0,                        /*    (b) */
               (FS_ERR   *)&err);


    switch (err) {
        case FS_ERR_NONE:
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
             return (DEF_FAIL);

        default:
             return (DEF_FAIL);
    }
                                                      /* (3)    */
    FSVol_Open((CPU_CHAR         *)"sdcard:0:",       /*    (a) */
               (CPU_CHAR         *)"sdcard:0:",       /*    (b) */
               (FS_PARTITION_NBR  ) 0,                /*    (c) */
               (FS_ERR           *)&err);
```

```
    switch (err) {
        case FS_ERR_NONE:
             APP_TRACE_DBG(("    ...opened volume (mounted).\r\n"));
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
        case FS_ERR_PARTITION_NOT_FOUND:
             APP_TRACE_DBG(("    ...opened device (unmounted).\r\n"));
             return (DEF_FAIL);

        default:
             APP_TRACE_DBG(("    ...opening volume failed w/err = %d.\r\n\r\n", err));
             return (DEF_FAIL);
    }

    return (DEF_OK);
}
```

L15-1(1) Register the SD/MMC CardMode device driver `FSDev_SD_Card`.

L15-1(2) `FSDev_Open()` opens/initializes a file system device. The parameters are the device name (1a) and a pointer to a device driver-specific configuration structure (1b). The device name (1a) is composed of a device driver name ("sdcard"), a single colon, an ASCII-formatted integer (the unit number) and another colon. Since the SD/MMC CardMode driver requires no configuration, the configuration structure (1b) should be passed a NULL pointer.

Since SD/MMC are often removable media, it is possible for the device to not be present when `FSDev_Open()` is called. The device will still be added to the file system and a volume opened on the (not yet present) device. When the volume is later accessed, the file system will attempt to refresh the device information and detect a file system (see Section 3.01.02 for more information).

L15-1(3) `FSVol_Open()` opens/mounts a volume. The parameters are the volume name (2a), the device name (2b) and the partition that will be opened (2c). There is no restriction on the volume name (2a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partition, then the partition number (2c) should be zero.

If the SD/MMC initialization succeeds, the file system will produce the trace output as shown in Figure 15-3 (if a sufficiently high trace level is configured). See Section D.09 about configuring the trace level.

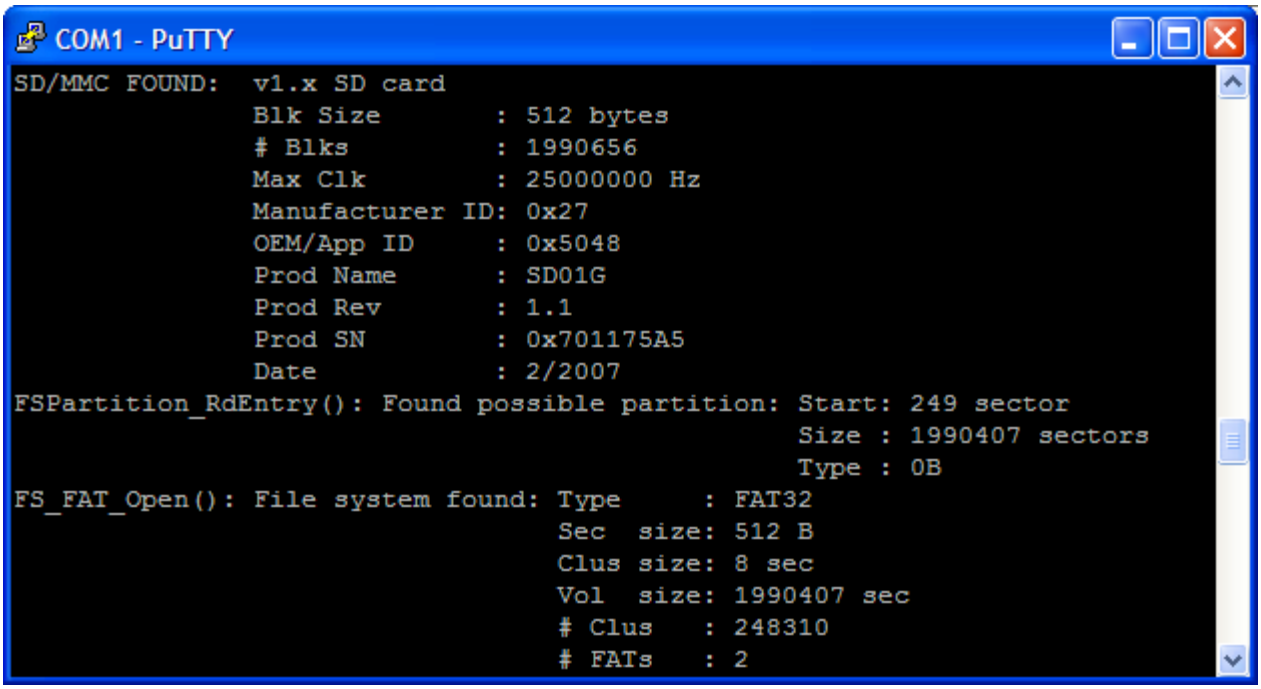


Figure 15-3. SD/MMC detection trace output.

15.02.01 SD/MMC CardMode Communication

In card mode, seven, nine or thirteen pins on the SD/MMC device are used, with the functions listed in the table below. All cards start up in "1 bit" mode (upon entering identification mode), which involves only a single data line. Once the host (the MCU/MPU) discovers the capabilities of the card, it may initiate 4- or 8-bit communication (the latter available only on new MMCs). Some card holders contain circuitry for card detect and write protect indicators, which the MCU/MPU may also monitor.

Pin	Name	Type	Description
1	CD/DAT3	I/O	Card Detect/Data Line (Bit 3)
2	CMD	I/O	Command/Response
3	V_{ss1}	S	Supply voltage ground
4	V_{DD}	S	Supply voltage
5	CLK	I	Clock
6	V_{SS2}	S	Supply voltage ground
7	DAT0	I/O	Data Line (Bit 0)
8	DAT1	I/O	Data Line (Bit 1)
9	DAT2	I/O	Data Line (Bit 2)
10	DAT4	I/O	Data Line (Bit 4)*
11	DAT5	I/O	Data Line (Bit 5)*
12	DAT6	I/O	Data Line (Bit 6)*
13	DAT7	I/O	Data Line (Bit 7)*

Table 15-2. SD/MMC pinout (Card Mode).
*Only present in MMC cards.

Exchanges between the host and card begin with a command (sent by the host on the CMD line), often followed by a response from the card (also on the CMD line); finally, one or more blocks data may be sent in one direction (on the data line(s)), each appended with a CRC.

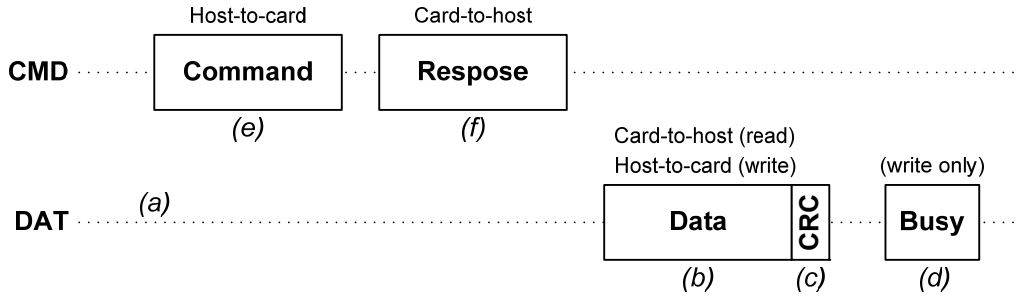


Figure 15-4. SD/MMC communication sequence.
(a) When no data is being transmitted, data lines are held low.
(b) Data block is preceded by a start bit ('0'); an end bit ('1') follows the CRC.
(c) The CRC is the 16-bit CCITT CRC.
(d) During the busy signaling following a write, DAT0 only is held low.
(e) See Figure 15-5 for description of the command format.
(f) See Figure 15-5 for description of the command format.

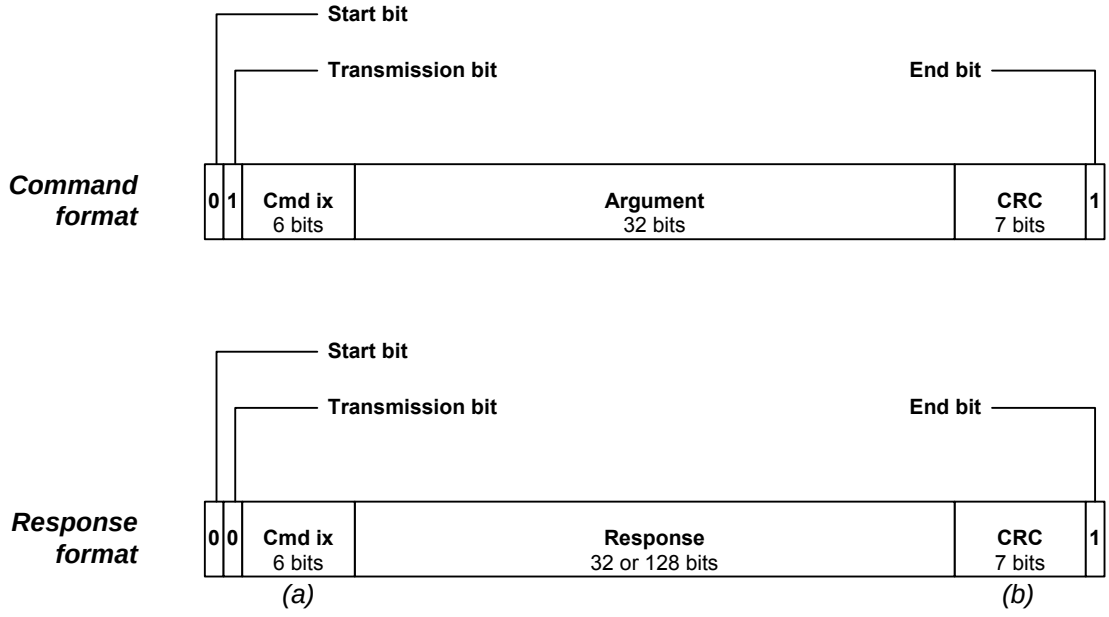


Figure 15-5. SD/MMC command and response formats.
(a) Command index is not valid for response formats R2 and R3.
(b) CRC is not valid for response format R3.

When a card is first connected to the host (at card power-on), it is in the 'inactive' state, awaiting a `GO_IDLE_STATE` command to start the initialization process, which is dependent on the card type. During initialization, the card starting in the 'idle' state moves through the 'ready' (as long as it supports the voltage range specified by the host) and 'identification' states (if it is assigned an address by or is assigned an address) before ending up in'standby'. It can now get selected by the host for data transfers. Figure 15-9 flowcharts this procedure.

15.02.02 SD/MMC CardMode Communication Debugging

The SD/MMC cardmode driver accesses the hardware through a port (BSP) as described in Section 15.03.03. A new BSP developed according to MCU/MPU documentation or by example must be verified step-by-step until flawless operation is achieved:

1. **Initialization (1-bit)**. Initialization must succeed for a SD/MMC card in 1-bit mode.
2. **Initialization (4- or 8-bit)**. Initialization must succeed for a SD/MMC card in 4 or 8-bit mode.
3. **Read data**. Data must be read from card, in both single- and multiple-block transactions.
4. **Write data**. Data must be written to the card, in both single and multiple-block transactions, and subsequently verified (by reading the modified sectors and comparing to the intended contents).

The (1-bit) initialization process reveals that commands can be executed and responses are returned with the proper bits in the correct byte-order. Example responses for each step in the sequence are given in Figures 15-10 and 15-11. The first command executed, `GO_IDLE_STATE`, never receives a response from the card. Only V2 SD cards respond to `SEND_IF_COND`, returning the check pattern sent to the card and the accepted voltage range. The OCR register, read with `SD_SEND_OP_COND` or `SEND_OP_COND`, assumes basically the same format for all card types. Finally, the CID (card ID) and CSD (card-specific data) registers are read—the only times 'long' (132-bit) responses are returned.

Multiple-bit initialization (often 4-bit) when performed on a SD card further confirms that the 8-byte SCR register and 64-byte SD status can be read and that the bus width can be set in the BSP. Though all current cards support 4-bit mode operation, the `SD_BUS_WIDTHS` field of the SCR is checked before configure the card bus width. Afterwards, the 64-byte SD status is read to see whether the bus width change was accomplished. When first debugging a port, it may be best to force multi-bit operation disabled by returning 1 from the BSP function `FSDev_SD_Card_BSP_GetBusWidthMax()`.

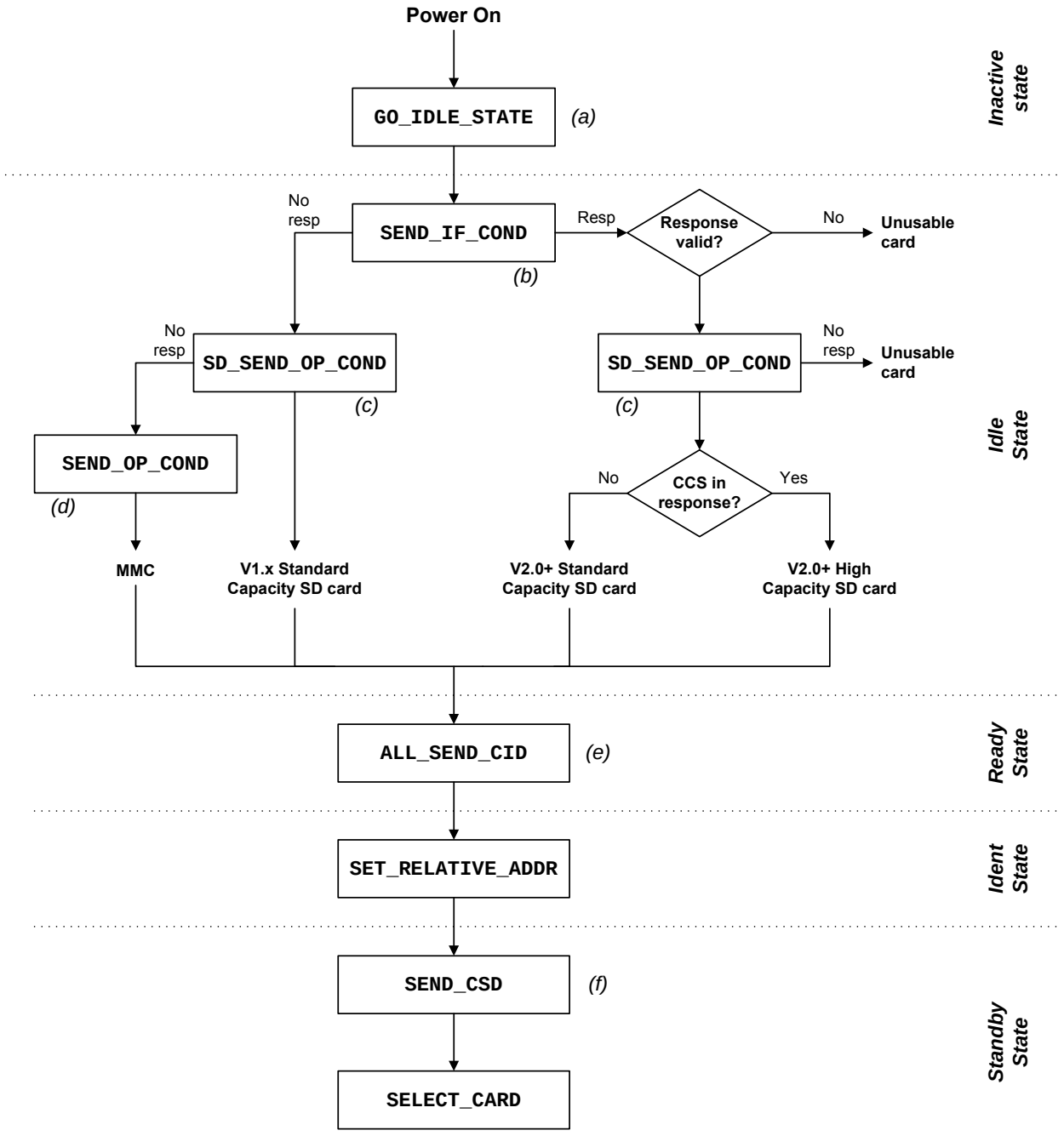


Figure 15-6. Simplified SD/MMC cardmode initialization and state transitions.

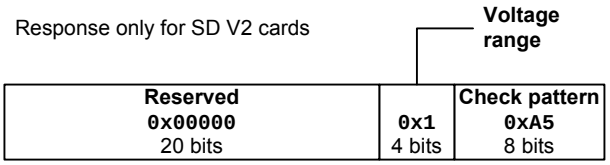
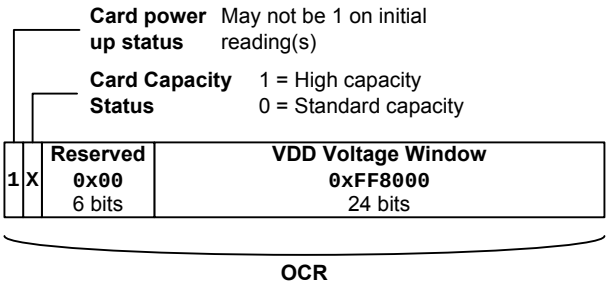
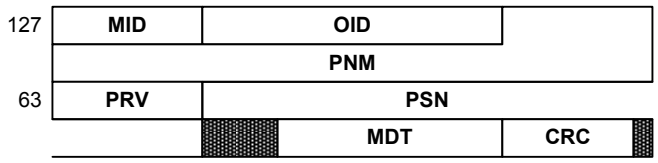
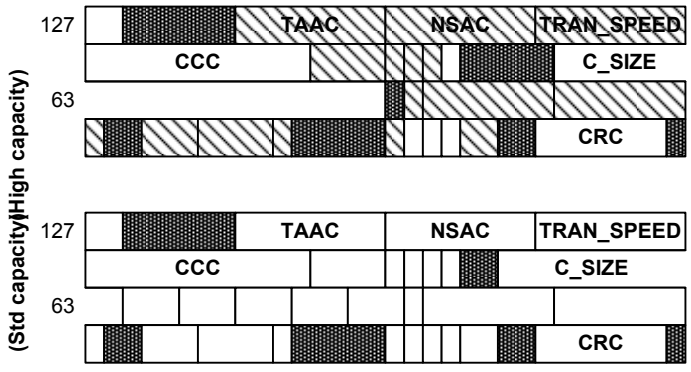
Command	Response
GO_IDLE_STATE Fig 15-6 (a)	No response
SEND_IF_COND Fig 15-6 (b)	
SD_SEND_OP_COND Fig 15-6 (c)	
ALL_SEND_CID Fig 15-6 (e)	 MID = Manufacturer ID = 0x03 OID = OEM/Application ID = 0x5344 PNM = Product name = 0x5344303247 = “SD02G” PRV = Product revision = 0x80 = 8.0 PSN = Product serial number = 0x021A7C83 MDT = Manufacturing date = 0x008 **Example** 0x03534453 0x44303247 0x80021A7C 0x83008B3A
SEND_CSD Fig 15-6 (f)	 **Examples** 0x400E0032 0x5B590000 0x1E5C7F80 0x0A4040DE 0x00260032 0x5F5A83C9 0x3EFBCFFF 0x928040CA

Figure 15-7. Command responses (SD card)

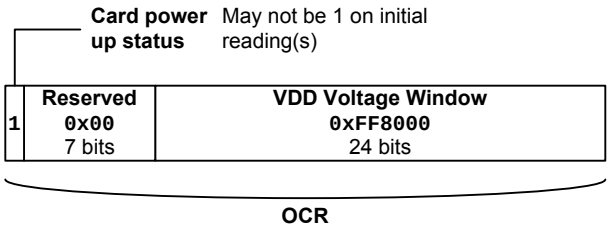
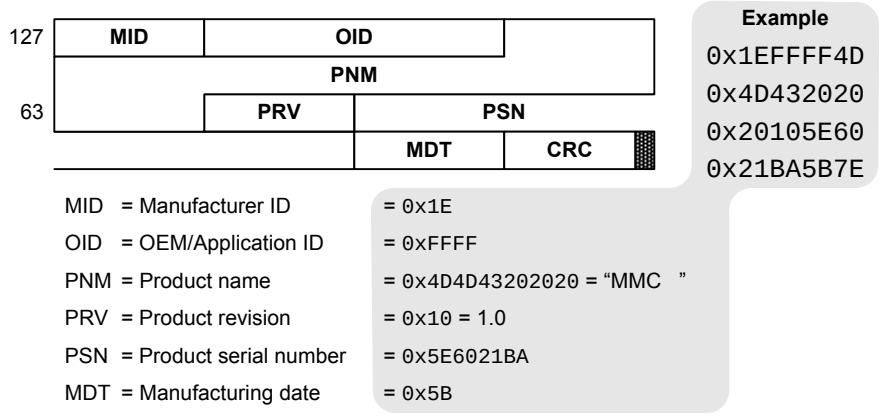
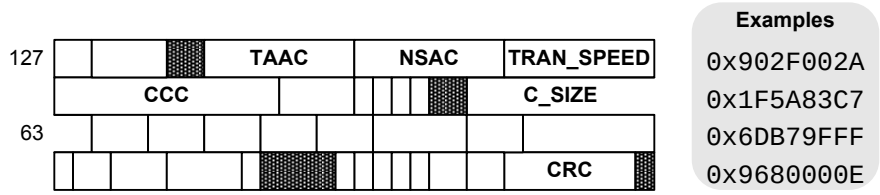
Command	Response
GO_IDLE_STATE Fig 15-6 (a)	No response
SEND_OP_COND Fig 15-6 (d)	
ALL_SEND_CID Fig 15-6 (e)	
SEND_CSD Fig 15-6 (f)	

Figure 15-8. Command responses (MMC card)

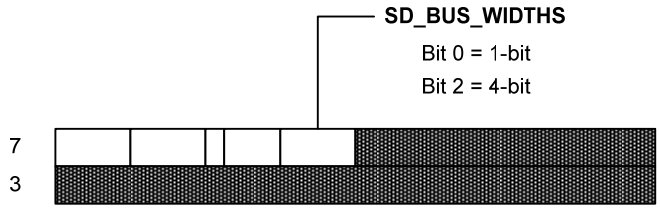


Figure 15-9. SD SCR Register

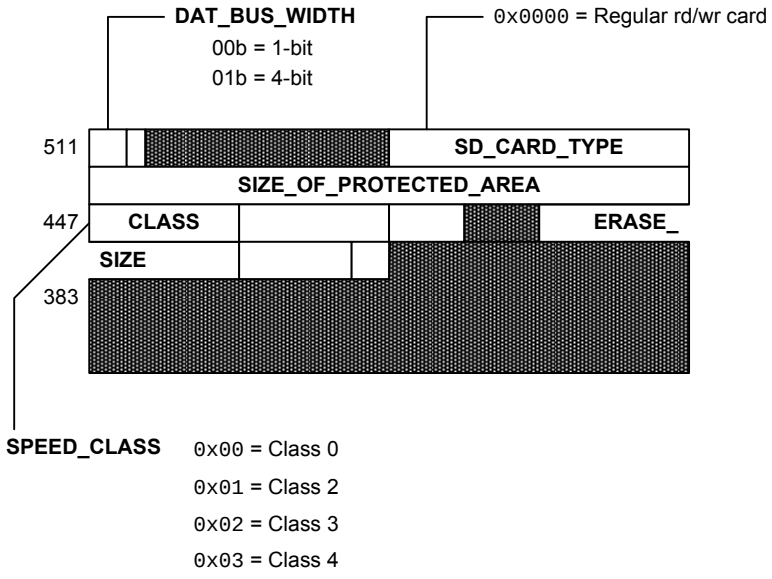


Figure 15-10. SD Status

15.02.03 SD/MMC CardMode BSP Overview

A BSP is required so that the SD/MMC cardmode driver will work on a particular system. The functions shown in the table below must be implemented.

Function	Description
`FSDev_SD_Card_BSP_Open()`	Open (initialize) SD/MMC card interface.
`FSDev_SD_Card_BSP_Close()`	Close (uninitialize) SD/MMC card interface.
`FSDev_SD_Card_BSP_Lock()`	Acquire SD/MMC card bus lock.
`FSDev_SD_Card_BSP_Unlock()`	Release SD/MMC card bus lock.
`FSDev_SD_Card_BSP_CmdStart()`	Start a command.
`FSDev_SD_Card_BSP_CmdWaitEnd()`	Wait for a command to end and get response.
`FSDev_SD_Card_BSP_CmdDataRd()`	Read data following command.
`FSDev_SD_Card_BSP_CmdDataWr()`	Write data following command.
`FSDev_SD_Card_BSP_GetBlkCntMax()`	Get max block count.
`FSDev_SD_Card_BSP_GetBusWidthMax()`	Get maximum bus width, in bits.
`FSDev_SD_Card_BSP_SetBusWidth()`	Set bus width.
`FSDev_SD_Card_BSP_SetClkFreq()`	Set clock frequency.
`FSDev_SD_Card_BSP_SetTimeoutData()`	Set data timeout.
`FSDev_SD_Card_BSP_SetTimeoutResp()`	Set response timeout

Table 15-3. SD/MMC cardmode BSP functions

The `Open()`/`Close()` functions are called upon open/close or medium change; these calls are always matched. The status and information functions (`GetBlkCntMax()`, `GetBusWidthMax()`, `SetBusWidth()`, `SetClkFreq()`, `SetTimeoutData()`, `SetTimeoutResp()`) help configure the new card upon insertion. `Lock()` and `Unlock()` surround all card accesses.

The remaining functions (`CmdStart()`, `CmdWaitEnd()`, `CmdDataRd()`, `CmdDataWr()`) constitute the command execution state machine (see Figure 15-11). A return error from one of the functions will abort the state machine, so the requisite considerations, such as preparing for the next command or preventing further interrupts, must be first handled.

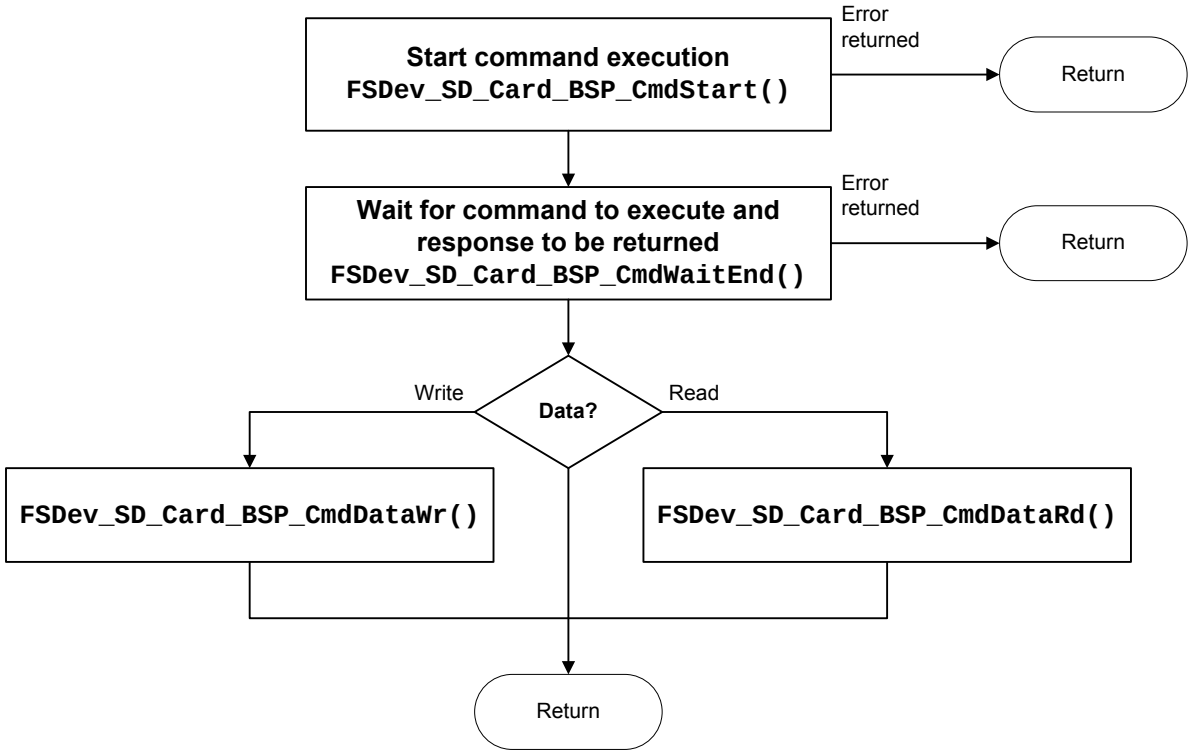


Figure 15-11. Command execution

15.02.03.01 Open (Initialize) Card Interface: FSDev_SD_Card_BSP_Open()

Open (initialize) SD/MMC card interface.

Prototype

```
CPU_BOOLEAN  FSDev_SD_Card_BSP_Open (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of SD/MMC card.

Returned Value

`DEF_OK`, if interface was opened.
`DEF_FAIL`, otherwise.

Notes/Warnings

This function will be called every time the device is opened.

15.02.03.02 Close (Uninitialize) Card Interface: `FSDev_SD_Card_BSP_Close()`

Close (uninitialize) SD/MMC card interface.

Prototype

```
void   FSDev_SD_Card_BSP_Close (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of SD/MMC card.

Returned Value

None.

Notes/Warnings

This function will be called every time the device is closed.

15.02.03.03 Acquire / Release SPI Lock: FSDev_SD_SPI_BSP_Lock()/Unlock()

Acquire/release SD/MMC card bus lock.

Prototypes

```
void  FSDev_SD_Card_BSP_Lock   (FS_QTY  unit_nbr);

void  FSDev_SD_Card_BSP_Unlock (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of SD/MMC card.

Returned Value

None.

Notes/Warnings

`FSDev_SD_SPI_BSP_Lock()` will be called before the SD/MMC driver begins to access the SD/MMC card bus. The application should NOT use the same bus to access another device until the matching call to `FSDev_SD_Card_BSP_Unlock()` has been made.

15.02.03.04 Start a Command: FSDev_SD_Card_BSP_CmdStart()

Start a command.

Prototype

```
void  FSDev_SD_Card_BSP_CmdStart (FS_QTY                unit_nbr,
                                  FS_DEV_SD_CARD_CMD   *p_cmd,
                                  void                 *p_data,
                                  FS_ERR               *p_err);
```

Arguments

`unit_nbr` Unit number of SD/MMC card.

`p_cmd` Pointer to command to transmit (see Note #2).

`P_data` Pointer to buffer address for DMA transfer (see Note #3).

`p_err` Pointer to variable t hat will receive the return erro code from this function:

`FS_DEV_SD_CARD_ERR_NONE`	No error.
`FS_DEV_SD_CARD_ERR_NO_CARD`	No card present.
`FS_DEV_SD_CARD_ERR_BUSY`	Controller is busy.
`FS_DEV_SD_CARD_ERR_UNKNOWN`	Unknown or other error.

Returned Value

None.

Notes/Warnings

1. The command start will be followed by zero, one or two additional BSP function calls, depending on whether data should be transferred and on whether any errors occur.

    a. `FSDev_SD_Card_BSP_CmdStart()` starts execution of the command. It may also set up the DMA transfer (if necessary).

    b. `FSDev_SD_Card_BSP_CmdWaitEnd()` waits for the execution of the command to end, getting the command response (if any).

    c. If data should be transferred from the card to the host, `FSDev_SD_Card_BSP_CmdDataRd()` will read that data; if data should be transferred from the host to the card, `FSDev_SD_Card_BSP_CmdDataWr()` will write that data.

    d. If an error is returned at any point, the sequence will be aborted.

2. The command `p_cmd` has the following parameters:

    a. `p_cmd->Cmd` is the command index.

b. `p_cmd->Arg` is the 32-bit argument (or 0 if there is no argument).

c. `p_cmd->Flags` is a bit-mapped variable with zero or more command flags:

`FS_DEV_SD_CARD_CMD_FLAG_INIT`	Initialization sequence before command
`FS_DEV_SD_CARD_CMD_FLAG_BUSY`	Busy signal expected after command
`FS_DEV_SD_CARD_CMD_FLAG_CRC_VALID`	CRC valid after command
`FS_DEV_SD_CARD_CMD_FLAG_IX_VALID`	Index valid after command
`FS_DEV_SD_CARD_CMD_FLAG_OPEN_DRAIN`	Command line is open drain
`FS_DEV_SD_CARD_CMD_FLAG_DATA_START`	Data start command
`FS_DEV_SD_CARD_CMD_FLAG_DATA_STOP`	Data stop command.
`FS_DEV_SD_CARD_CMD_FLAG_RESP`	Response expected
`FS_DEV_SD_CARD_CMD_FLAG_RESP_LONG`	Long response expected

d. `p_cmd->DataDir` indicates the direction of any data transfer that should follow this command, if any:

`FS_DEV_SD_CARD_DATA_DIR_NONE`	No data transfer
`FS_DEV_SD_CARD_DATA_DIR_HOST_TO_CARD`	Transfer host-to-card (write)
`FS_DEV_SD_CARD_DATA_DIR_CARD_TO_HOST`	Transfer card-to-host (read)

e. `p_cmd->DataType` indicates the type of the data transfer that should follow this command, if any:

`FS_DEV_SD_CARD_DATA_TYPE_NONE`	No data transfer
`FS_DEV_SD_CARD_DATA_TYPE_SINGLE_BLOCK`	Single data block
`FS_DEV_SD_CARD_DATA_TYPE_MULTI_BLOCK`	Multiple data blocks
`FS_DEV_SD_CARD_DATA_TYPE_STREAM`	Stream data

f. `p_cmd->RespType` indicates the type of the response that should be expected from this command:

`FS_DEV_SD_CARD_RESP_TYPE_NONE`	No response
`FS_DEV_SD_CARD_RESP_TYPE_R1`	R1 response
`FS_DEV_SD_CARD_RESP_TYPE_R1B`	R1b response
`FS_DEV_SD_CARD_RESP_TYPE_R2`	R2 response
`FS_DEV_SD_CARD_RESP_TYPE_R3`	R3 response
`FS_DEV_SD_CARD_RESP_TYPE_R4`	R4 response
`FS_DEV_SD_CARD_RESP_TYPE_R5`	R5 response
`FS_DEV_SD_CARD_RESP_TYPE_R5B`	R5B response
`FS_DEV_SD_CARD_RESP_TYPE_R6`	R6 response
`FS_DEV_SD_CARD_RESP_TYPE_R7`	R7 response

g. `p_cmd->BlkSize` and `p_cmd->BlkCnt` are the block size and block count of the data transfer that should follow this command, if any.

3. The pointer to the data buffer that will receive the data transfer that should follow this command is given so that a DMA transfer can be set up.

15.02.03.05 Wait for Command to End: FSDev_SD_Card_BSP_CmdWaitEnd()

Wait for command to end and get command response.

Prototype

```
void  FSDev_SD_Card_BSP_CmdWaitEnd (FS_QTY                unit_nbr,
                                    FS_DEV_SD_CARD_CMD   *p_cmd,
                                    CPU_INT32U           *p_resp,
                                    FS_ERR               *p_err);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

`p_cmd` Pointer to command that is ending

`P_data` Pointer to buffer that will receive command response, if any

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_DEV_SD_CARD_ERR_NONE`	No error
`FS_DEV_SD_CARD_ERR_NO_CARD`	No card present
`FS_DEV_SD_CARD_ERR_UNKNOWN`	Unknown or other error
`FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT`	Timeout in waiting for command response
`FS_DEV_SD_CARD_ERR_RESP_TIMEOUT`	Timeout in receiving command response
`FS_DEV_SD_CARD_ERR_RESP_CHKSUM`	Error in response checksum
`FS_DEV_SD_CARD_ERR_RESP_CMD_IX`	Response command index error
`FS_DEV_SD_CARD_ERR_RESP_END_BIT`	Response end bit error
`FS_DEV_SD_CARD_ERR_RESP`	Other response error
`FS_DEV_SD_CARD_ERR_DATA`	Other data err

Returned Value

None.

Notes/Warnings

1. This function will be called even if no response is expected from the command.

2. This function will NOT be called if `FSDev_SD_Card_BSP_CmdStart()` returned an error.

3. For a command with a normal response, a 4-byte response should be stored in `p_resp`.

4. For a command with a long response, a 16-byte response should be stored in `p_resp`.
   The first 4-byte word should hold bits 127..96 of the response.
   The second 4-byte word should hold bits 95..64 of the response.
   The third 4-byte word should hold bits 63..32 of the response.
   The fourth 4-byte word should hold bits 31.. 0 of the response.

15.02.03.06 Read Data following Command: FSDev_SD_Card_BSP_CmdDataRd()

Read data following command.

Prototype

```
void  FSDev_SD_Card_BSP_CmdDataRd (FS_QTY                unit_nbr,
                                   FS_DEV_SD_CARD_CMD   *p_cmd,
                                   void                 *p_dest,
                                   FS_ERR               *p_err);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

`p_cmd` Pointer to command that was started

`p_dest` Pointer to destination buffer

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_DEV_SD_CARD_ERR_NONE`	No error
`FS_DEV_SD_CARD_ERR_NO_CARD`	No card present
`FS_DEV_SD_CARD_ERR_UNKNOWN`	Unknown or other error
`FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT`	Timeout in waiting for data
`FS_DEV_SD_CARD_ERR_DATA_OVERRUN`	Data overrun
`FS_DEV_SD_CARD_ERR_DATA_TIMEOUT`	Timeout in receiving data
`FS_DEV_SD_CARD_ERR_DATA_CHKSUM`	Error in data checksum
`FS_DEV_SD_CARD_ERR_DATA_START_BIT`	Data start bit error
`FS_DEV_SD_CARD_ERR_DATA`	Other data err

Returned Value

None.

Notes/Warnings

None.

15.02.03.07 Write Data following Command: FSDev_SD_Card_BSP_CmdDataWr()

Write data following command.

Prototype

```
void  FSDev_SD_Card_BSP_CmdDataWr (FS_QTY                unit_nbr,
                                   FS_DEV_SD_CARD_CMD   *p_cmd,
                                   void                 *p_src,
                                   FS_ERR               *p_err);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

`p_cmd` Pointer to command that was started

`p_src` Pointer to source buffer

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_DEV_SD_CARD_ERR_NONE`	No error
`FS_DEV_SD_CARD_ERR_NO_CARD`	No card present
`FS_DEV_SD_CARD_ERR_UNKNOWN`	Unknown or other error
`FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT`	Timeout in waiting for data
`FS_DEV_SD_CARD_ERR_DATA_UNDERRUN`	Data underrun
`FS_DEV_SD_CARD_ERR_DATA_CHKSUM`	Error in data checksum
`FS_DEV_SD_CARD_ERR_DATA_START_BIT`	Data start bit error
`FS_DEV_SD_CARD_ERR_DATA`	Other data err

Returned Value

None.

Notes/Warnings

None.

15.02.03.08 Get Maximum Block Count: FSDev_SD_Card_BSP_GetBlkCntMax()

Get maximum number of blocks that can be transferred with a multiple read or multiple write command.

Prototype

```
CPU_INT32U  FSDev_SD_Card_BSP_GetBlkCntMax (FS_QTY      unit_nbr,
                                            CPU_INT32U  blk_size);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

`blk_size` Block size, in octets.

Returned Value

Maximum number of blocks

Notes/Warnings

1. The DMA region from which data is read or written may be a limited size. The count returned by this function should be the number of blocks of size `blk_size` that can fit into this region.

2. If the controller is not capable of multiple block reads or writes, 1 should be returned.

3. If the controller has no limit on the number of blocks in a multiple block read or write, `DEF_INT_32U_MAX_VAL` should be returned.

4. This function SHOULD always return the same value. If hardware constraints change at run-time, the device MUST be closed & re-opened for any changes to be effective.

15.02.03.09 Get Maximum Bus Width: FSDev_SD_Card_BSP_GetBusWidthMax()

Get maximum bus width, in bits.

Prototype

```
CPU_INT08U  FSDev_SD_Card_BSP_GetBusWidthMax (FS_QTY  unit_nbr);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

Returned Value

Maximum bus width

Notes/Warnings

1. Legal values are typically 1, 4 & 8.
2. This function SHOULD always return the same value. If hardware constraints change at run-time, the device MUST be closed & re-opened for any changes to be effective.

15.02.03.10 Set Bus Width: FSDev_SD_Card_BSP_SetBusWidth()

Set bus width.

Prototype

```
void  FSDev_SD_Card_BSP_SetBusWidth (FS_QTY      unit_nbr,
                                     CPU_INT08U  width);
```

Arguments

`unit_nbr`	Unit number of SD/MMC card
`width`	Bus width, in bits

Returned Value

None

Notes/Warnings

None

15.02.03.11 Set Clock Frequency: FSDev_SD_Card_BSP_SetClkFreq()

Set clock frequency.

Prototype

```
void  FSDev_SD_Card_BSP_SetClkFreq (FS_QTY      unit_nbr,
                                    CPU_INT32U  freq);
```

Arguments

`unit_nbr`	Unit number of SD/MMC card
`freq`	Clock frequency, in Hz

Returned Value

None

Notes/Warnings

The effective clock frequency MUST be no more than `freq`. If the frequency cannot be configured equal to `freq`, it should be configured less than `freq`.

15.02.03.12 Set Data Timeout: FSDev_SD_Card_BSP_SetDataTimeout()

Set data timeout.

Prototype

```
void  FSDev_SD_Card_BSP_SetDataTimeout (FS_QTY      unit_nbr,
                                        CPU_INT32U  to_clks);
```

Arguments

`unit_nbr` Unit number of SD/MMC card

`to_clks` Timeout, in clocks

Returned Value

None

Notes/Warnings

None

15.02.03.13 Set Response Timeout: FSDev_SD_Card_BSP_SetRespTimeout()

Set response timeout.

Prototype

```
void  FSDev_SD_Card_BSP_SetRespTimeout (FS_QTY      unit_nbr,
                                        CPU_INT32U  to_ms);
```

Arguments

`unit_nbr`	Unit number of SD/MMC card
`to_ms`	Timeout, in milliseconds

Returned Value

None

Notes/Warnings

None

15.03 Using the SD/MMC SPI Driver

To use the SD/MMC SPI driver, five files, in addition to the generic file system files, must be included in the build:

- *fs_dev_sd.c* (located in the directory specified in Section 15.01).
- *fs_dev_sd.h* (located in the directory specified in Section 15.01).
- *fs_dev_sd_spi.c* (located in the directory specified in Section 15.01).
- *fs_dev_sd_spi.h* (located in the directory specified in Section 15.01).
- *fs_dev_sd_spi_bsp.c* (located in the user application or BSP).

The file *fs_dev_sd_spi.h* must also be `#included` in any application or header files that directly reference the driver (for example, by registering the device driver). The following directories must be on the project include path:

- *\Micrium\Software\uC-FS\Dev\SD*
- *\Micrium\Software\uC-FS\Dev\SD\SPI*

A single SD/MMC volume is opened as shown in Listing 15-2. The file system initialization (`FS_Init()`) function must have previously been called.

ROM/RAM characteristics and performance benchmarks of the SD/MMC driver can be found in Section 8.01.01. The SD/MMC driver also provides interface functions to get low-level card information and read the Card ID and Card-Specific Data registers (see Section A.11).

Listing 15-2. Opening a SD/MMC device volume.

```
FS_ERR  App_FS_AddSD_SPI (void)
{
    FS_ERR       err;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_SD_SPI,        /* (1)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) && (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        return (DEF_FAIL);
    }
                                                     /* (2)    */
    FSDev_Open((CPU_CHAR *)"sd:0:",                  /*    (a) */
               (void     *) 0,                       /*    (b) */
               (FS_ERR   *)&err);


    switch (err) {
        case FS_ERR_NONE:
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
             return (DEF_FAIL);

        default:
             return (DEF_FAIL);
    }
                                                    /* (3)    */
    FSVol_Open((CPU_CHAR         *)"sd:0:",         /*    (a) */
               (CPU_CHAR         *)"sd:0:",         /*    (b) */
               (FS_PARTITION_NBR  ) 0,              /*    (c) */
               (FS_ERR           *)&err);


    switch (err) {
```

```
        case FS_ERR_NONE:
             APP_TRACE_DBG(("    ...opened volume (mounted).\r\n"));
             break;

        case FS_ERR_DEV:
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
        case FS_ERR_PARTITION_NOT_FOUND:
             APP_TRACE_DBG(("    ...opened device (unmounted).\r\n"));
             return (DEF_FAIL);

        default:
             APP_TRACE_DBG(("    ...opening volume failed w/err = %d.\r\n\r\n", err));
             return (DEF_FAIL);
    }

    return (DEF_OK);

}
```

L15-2(1) Register the SD/MMC SPI device driver `FSDev_SD_SPI`.

L15-2(2) `FSDev_Open()` opens/initializes a file system device. The parameters are the device name (1a) and a pointer to a device driver-specific configuration structure (1b). The device name (1a) is composed of a device driver name ("sd"), a single colon, an ASCII-formatted integer (the unit number) and another colon. Since the SD/MMC SPI driver requires no configuration, the configuration structure (1b) should be passed a NULL pointer.

Since SD/MMC are often removable media, it is possible for the device to not be present when `FSDev_Open()` is called. The device will still be added to the file system and a volume opened on the (not yet present) device. When the volume is later accessed, the file system will attempt to refresh the device information and detect a file system (see Section 3.01.02 for more information).

L15-2(3) `FSVol_Open()` opens/mounts a volume. The parameters are the volume name (2a), the device name (2b) and the partition that will be opened (2c). There is no restriction on the volume name (2a); however, it is typical to give the volume the same name as the underlying device. If the default partition is to be opened, or if the device is not partition, then the partition number (2c) should be zero.

If the SD/MMC initialization succeeds, the file system will produce the trace output as shown in Figure 15-12 (if a sufficiently high trace level is configured). See Section D.09 about configuring the trace level.

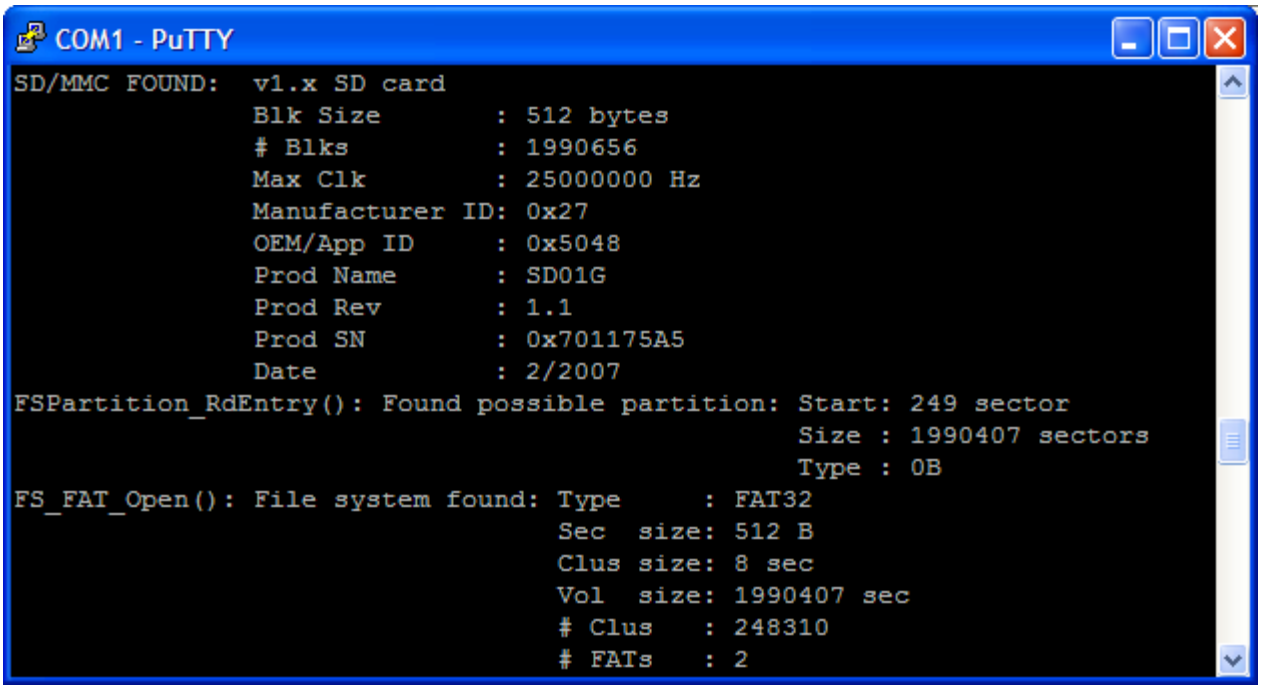


Figure 15-12. SD/MMC detection trace output.

15.03.01 SD/MMC SPI Communication

SPI is a simple protocol supported by peripherals commonly built-in on CPUs. Moreover, since the communication can easily be accomplished by software control of GPIO pins ("software SPI" or "bit-banging"), a SD/MMC card can be connected to almost any platform. In SPI mode, seven pins on the SD/MMC device are used, with the functions listed in Table 15-3. As with any SPI device, four signals are used to communicate with the host (CS, DataIn, CLK and DataOut). Some card holders contain circuitry for card detect and write protect indicators, which the MCU/MPU may also monitor.

Pin	Name	Type	Description
1	CS	I	Chip Select
2	DataIn	I	Host-to-card commands and data
3	V_{ss1}	S	Supply voltage ground
4	V_{DD}	S	Supply voltage
5	CLK	I	Clock
6	V_{SS2}	S	Supply voltage ground
7	DataOut	O	Card-to-host data and status

Table 15-4. SD/MMC Pinout (SPI Mode).

The four signals connecting the host (or master) and card (also known as the slave) are named variously in different manuals and documents. The DataIn pin of the card is also known as MOSI (Master Out Slave In); it is the data output of the host CPU. Similarly, the DataOut pin of the card is also known as MISO (Master In Slave Out); it is the data input of the host CPU. The CS and CLK pins (also known as SSEL and SCK) are the chip select and clock pins. The host selects the slave by asserting CS, potentially allowing it to choose a single peripheral among several that are sharing the bus (i.e., by sharing the CLK, MOSI and MISO signals).

When a card is first connected to the host (at card power-on), it is in the 'inactive' state, awaiting a `GO_IDLE_STATE` command to start the initialization process. The card will enter SPI mode (rather than card mode) because the driver holds the CS signal low while executing the `GO_IDLE_STATE` command. The card now in the 'idle' state moves through the 'ready' (as long as it supports the voltage range specified by the host) before ending up in'standby'. It can now get selected by the host (using the chip select) for data transfers. Figure 15-5 flowcharts this procedure.

15.03.02 SD/MMC SPI Communication Debugging

The SD/MMC SPI driver accesses the hardware through a port (SPI BSP) as described in Section 15.02.03. A new BSP developed according to MCU/MPU documentation or by example must be verified step-by-step until flawless operation is achieved:

5. **Initialization**. Initialization must succeed.
6. **Read data**. Data must be read from card, in both single- and multiple-block transactions.
7. **Write data**. Data must be written to the card, in both single and multiple-block transactions, and subsequently verified (by reading the modified sectors and comparing to the intended contents).

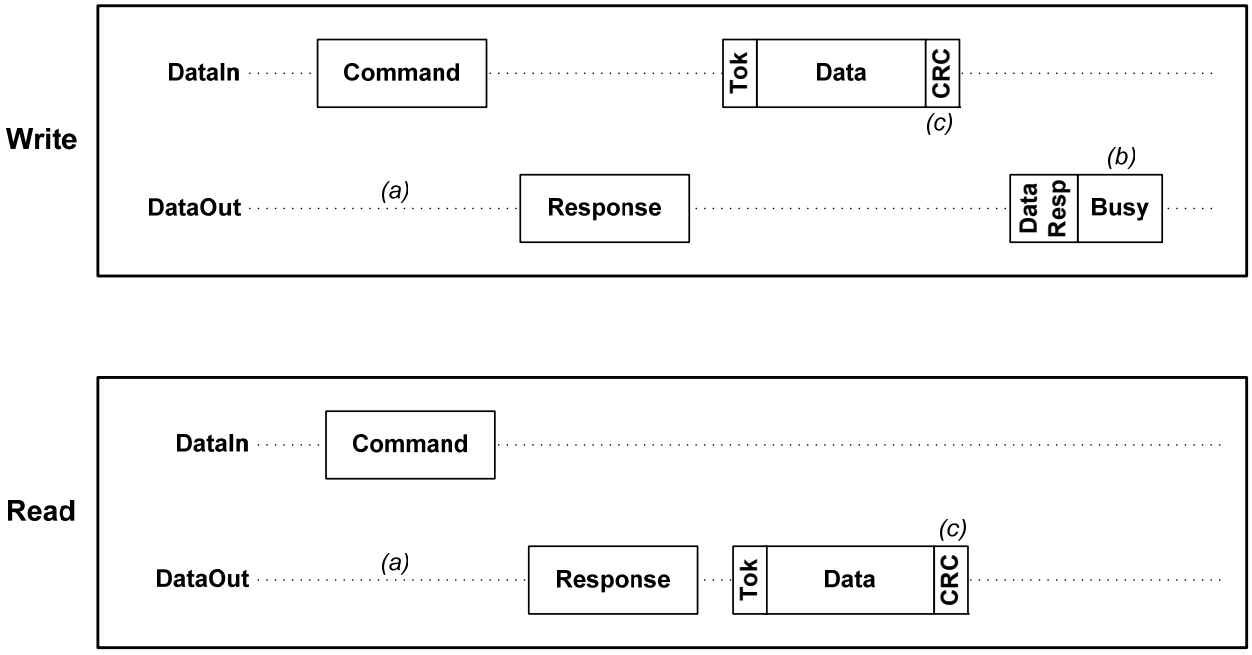


Figure 15-13. SD/MMC SPI mode communication sequence.
(a) When no data is being transmitted, DataOut line is held high.
(b) During busy signaling, DataOut line is held low.
(c) The CRC is the 16-bit CCITT CRC. By default, this is optional and dummy bytes may be transmitted instead. The card only checks the CRC if `CRC_ON_OFF` has been executed.

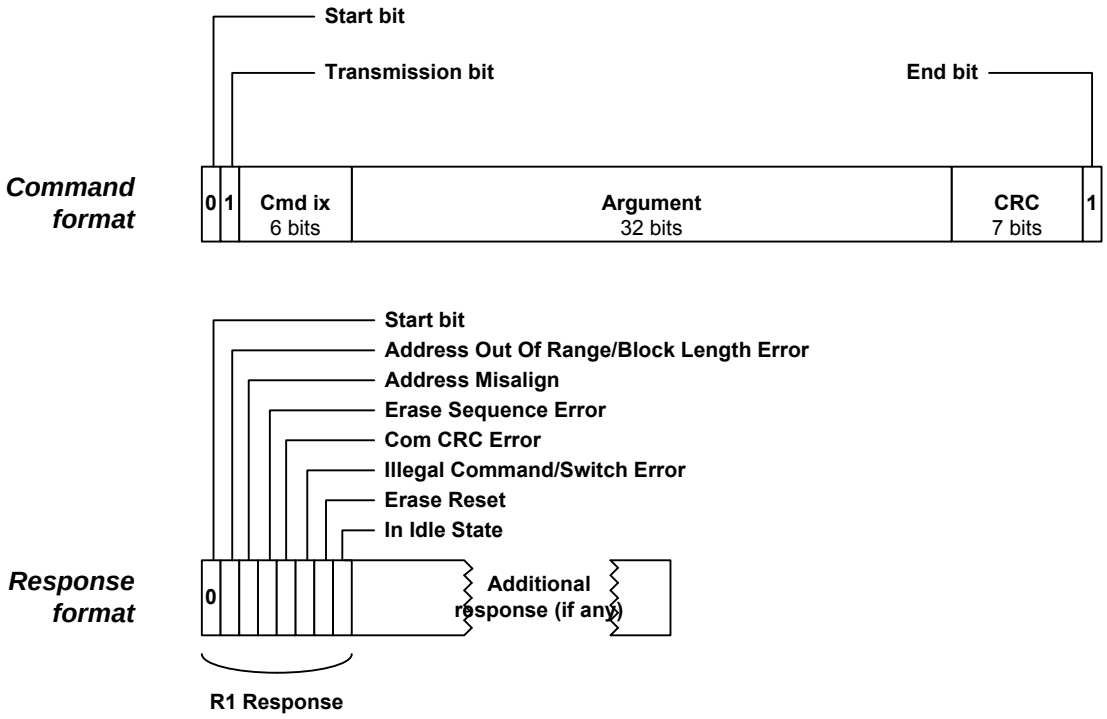


Figure 15-14. SD/MMC SPI mode command and response formats.

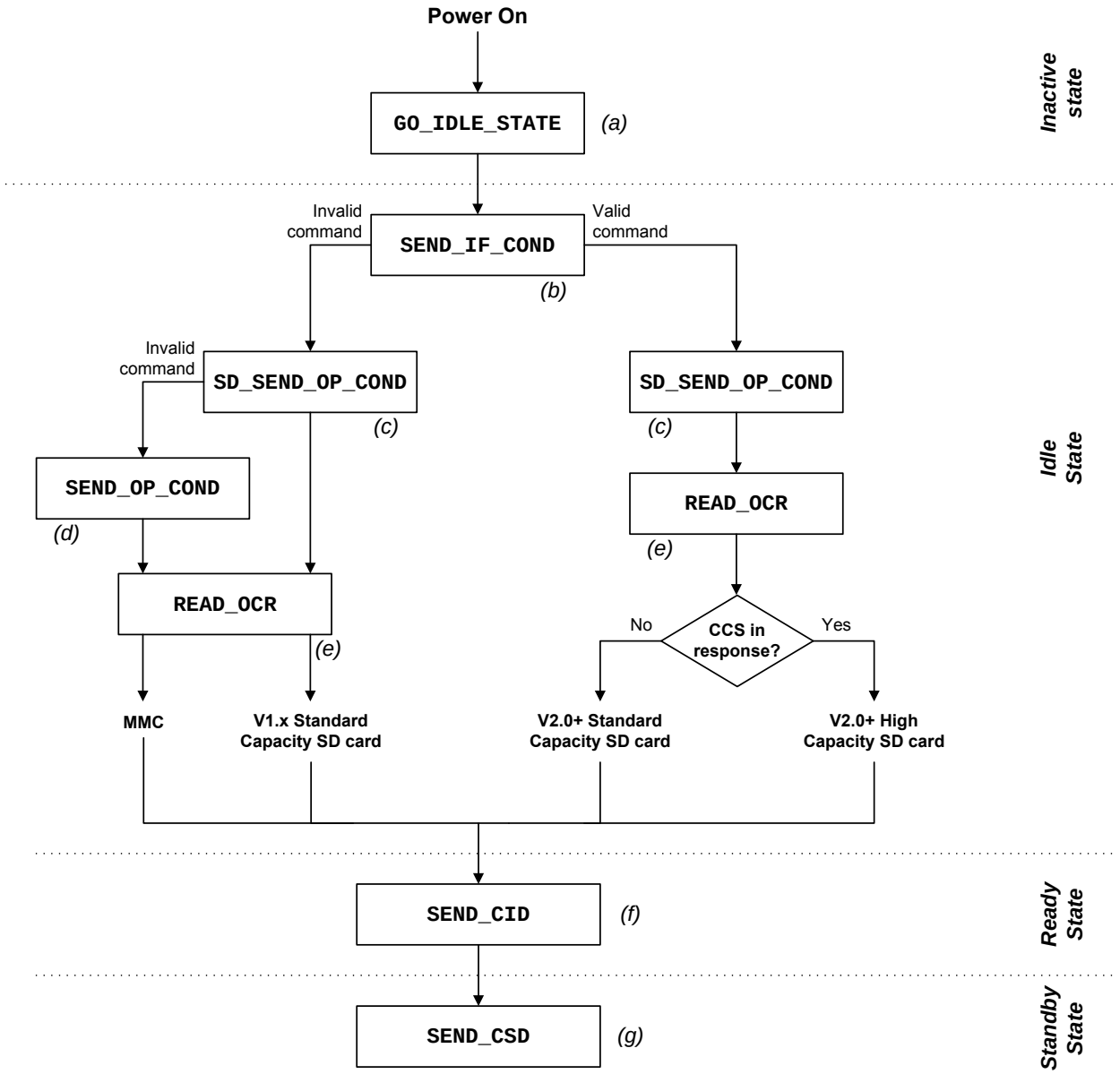


Figure 15-15. Simplified SD/MMC SPI mode initialization and state transitions.

The initialization process reveals that commands can be executed and proper responses are returned. The command responses in SPI mode are identical to those in cardmode (see Figures 15-10 and 15-11), except each is preceded by a R1 status byte. Obvious errors, such as improper initialization or failed chip select manipulation, will typically be caught here. More subtle conditions may appear intermittently during reading or writing.

15.03.03 SD/MMC SPI BSP Overview

An SPI BSP is required so that the SD/MMC SPI driver will work on a particular system. For more information about these functions, see Section 8.03.

Function	Description
`FSDev_SD_SPI_BSP_SPI_Open()`	Open (initialize) SPI.
`FSDev_SD_SPI_BSP_SPI_Close()`	Close (uninitialize) SPI.
`FSDev_SD_SPI_BSP_SPI_Lock()`	Acquire SPI lock.
`FSDev_SD_SPI_BSP_SPI_Unlock()`	Release SPI lock.
`FSDev_SD_SPI_BSP_SPI_Rd()`	Read from SPI.
`FSDev_SD_SPI_BSP_SPI_Wr()`	Write to SPI.
`FSDev_SD_SPI_BSP_SPI_ChipSelEn()`	Enable chip select.
`FSDev_SD_SPI_BSP_SPI_ChipSelDis()`	Disable chip select.
`FSDev_SD_SPI_BSP_SPI_SetClkFreq()`	Set SPI clock frequency.

Table 15-5. SD/MMC SPI BSP Functions

Appendix

A

µC/FS API Reference Manual

This chapter provides a reference to µC/FS services. The following information is provided for each entry:

- A brief description of the service
- The function prototype
- The filename of the source code
- The #define constant required to enable code for the service
- A description of the arguments passed to the function
- A description of returned value(s)
- Specific notes and warnings regarding use of the service
- One or two examples of how to use the function

Many functions return error codes. These error codes should be checked by the application to ensure that the µC/FS function performed its operation as expected.

Each of the user-accessible file system services is presented in alphabetical order within an appropriate section; the section for a particular function can be determined from its name.

Section	Functions begin with…
General file system functions	`FS_`
POXIX API functions	`fs_`
Device functions	`FSDev_`
Directory functions	`FSDir_`
Entry functions	`FSEntry_`
File functions	`FSFile_`
Time functions	`FSTime_`
Volume functions	`FSVol_`
NOR driver functions	`FSDev_NOR_`
SD/MMC driver functions	`FSDev_SD_`
Compact Flash/IDE driver functions	`FSDev_IDE_`
MSC driver functions	`FSDev_MSC_`
RAMDisk driver functions	`FSDev_RAM_`
FAT functions	`FS_FAT_`
BSP functions	`FS_BSP_`
OS functions	`FS_OS_`

A.01 GENERAL FILE SYSTEM FUNCTIONS

```
void
FS_DevDrvAdd      (FS_DEV_API   *p_dev_api,
                   FS_ERR       *p_err);
```

```
FS_ERR
FS_Init           (FS_CFG       *p_fs_cfg);
```

```
CPU_INT08U
FS_VersionGet     (void);
```

```
void
FS_WorkingDirGet  (CPU_CHAR     *path_dir,
                   CPU_SIZE_T    len_max,
                   FS_ERR       *p_err);
```

```
void
FS_WorkingDirSet  (CPU_CHAR     *path_dir,
                   FS_ERR       *p_err);
```

FS_DevDrvAdd()

```
void  FS_DevDrvAdd (FS_DEV_API  *p_dev_drv,
                    FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs.c	Application	N/A

Adds a device driver to the file system.

Arguments

`p_dev_drv` Pointer to device driver (see Section C.08).

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Device driver added.
`FS_ERR_NULL_PTR`	Argument `p_dev_drv` passed a NULL pointer.
`FS_ERR_DEV_DRV_ALREADY_ADDED`	Device driver already added.
`FS_ERR_DEV_DRV_INVALID_NAME`	Device driver name invalid.
`FS_ERR_DEV_DRV_NO_TBL_POS_AVAIL`	No device driver table position available.

Returned Value

None.

Notes/Warnings

1. The `NameGet()` device driver interface function **MUST** return a valid name:
    a. The name must be unique (e.g., a name that is not returned by any other device driver);
    b. The name must NOT include any of the characters: '`:`', '`\`' or '`/`'.
    c. The name must contain fewer than `FS_CFG_MAX_DEV_DRV_NAME_LEN` characters;
    d. The name must NOT be an empty string.

2. The `Init()` device driver interface function is called to initialize driver structures and any hardware for detecting the presence of devices (for a removable medium).

FS_Init()

```
FS_ERR  FS_Init (FS_CFG  *p_fs_cfg);
```

File	Called from	Code enabled by
fs.h	Application	N/A

Initializes µC/FS and MUST be called prior to calling any other µC/FS API functions.

Arguments

`p_fs_cfg` Pointer to file system configuration (see Section C.01).

Returned Value

FS_ERR_NONE, if successful;
Specific initialization error code, otherwise.

The return value **SHOULD** be inspected to determine whether µC/FS is successfully initialized or not. If µ/FS did **NOT** successfully initialize, search for the returned error in *fs_err.h* and source files to locate where µC/FS initialization failed.

Notes/Warnings

µC/LIB memory management function `Mem_Init()` **MUST** be called prior to calling this function.

FS_VersionGet()

```
CPU_INT16U  FS_VersionGet (void);
```

File	Called from	Code enabled by
fs.c	Application	N/A

Gets the µC/FS software version.

Arguments
None.

Returned Value
µC/FS software version.

Notes/Warnings
The value returned is multiplied by 100. For example, version 4.03 would be returned as 403.

FS_WorkingDirGet()

```
void  FS_WorkingDirGet (CPU_CHAR    *path_dir,
                        CPU_SIZE_T   size,
                        FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs.c	Application; `fs_getcwd()`	`FS_CFG_WORKING_DIR_EN`

Get the working directory for the current task.

Arguments

`path_dir` String buffer that will receive the working directory path.

`size` Size of string buffer.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Working directory obtained.
`FS_ERR_NULL_PTR`	Argument `path_dir` passed a NULL pointer.
`FS_ERR_NULL_ARG`	Argument `size` passed a NULL value.
`FS_ERR_NAME_BUF_TOO_SHORT`	Argument `size` less than length of path
`FS_ERR_VOL_NONE_EXIST`	No volumes exist.

Returned Value

None.

Notes/Warnings

If no working directory is assigned for the task, the default working directory—the root directory on the default volume—will be returned in the user buffer and set as the task's working directory.

FS_WorkingDirSet()

```
void  FS_WorkingDirSet (CPU_CHAR  *path_dir,
                        FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs.c	Application; `fs_chdir()`	`FS_CFG_WORKING_DIR_EN`

Set the working directory for the current task.

Arguments

`path_dir` String buffer that specified EITHER...

(a) the absolute working directory path to set;

(b) a relative path that will be applied to the current working directory.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Working directory set.
`FS_ERR_NULL_PTR`	Argument `path_dir` passed a NULL pointer.
`FS_ERR_VOL_NONE_EXIST`	No volumes exist.
`FS_ERR_WORKING_DIR_NONE_AVAIL`	No working directories available.
`FS_ERR_WORKING_DIR_INVALID`	Argument `path_dir` passed an invalid directory.

Returned Value

None.

Notes/Warnings

None.

A.02 POSIX API FUNCTIONS

```
char *
fs_asctime_r    (const  struct  fs_tm  *p_time,
                 char                  *str_time);
```

```
int
fs_chdir        (const  char           *path_dir);
```

```
void
fs_clearerr     (FS_FILE               *p_file);
```

```
int
fs_closedir     (FS_DIR                *p_dir);
```

```
char *
fs_ctime_r      (const  fs_time_t      *p_ts,
                 char                  *str_time);
```

```
int
fs_fclose       (FS_FILE               *p_file);
```

```
int
fs_feof         (FS_FILE               *p_file);
```

```
int
fs_ferror       (FS_FILE               *p_file);
```

```
int
fs_fflush       (FS_FILE               *p_file);
```

```
int
fs_fgetpos      (FS_FILE               *p_file,
                 fs_fpos_t             *p_pos);
```

```
void
fs_flockfile    (FS_FILE               *p_file);
```

```
FS_FILE  *
fs_fopen        (const  char           *name_full,
                 const  char           *str_mode);
```

```
fs_size_t
fs_fread        (void                  *p_dest,
                 fs_size_t              size,
                 fs_size_t              nitems,
                 FS_FILE               *p_file);
```

```
int
fs_fseek        (FS_FILE               *p_file,
                 long  int              offset,
                 int                    origin);
```

```
int
fs_fsetpos      (FS_FILE               *p_file,
                 fs_fpos_t             *p_pos);
```

```
long  int
fs_ftell        (FS_FILE               *p_file);
```

```
int
fs_ftruncate    (FS_FILE                *p_file,
                 fs_off_t                size);
```

```
int
fs_ftrylockfile (FS_FILE                *p_file);
```

```
void
fs_funlockfile  (FS_FILE                *p_file);
```

```
fs_size_t
fs_fwrite       (void                   *p_src,
                 fs_size_t               size,
                 fs_size_t               nitems,
                 FS_FILE                *p_file);
```

```
char  *
fs_getcwd       (char                   *path_dir,
                 fs_size_t               size);
```

```
struct  fs_tm  *
fs_localtime_r  (const  fs_time_t      *p_ts,
                 struct  fs_tm         *p_time);
```

```
int
fs_mkdir        (const  char           *name_full);
```

```
fs_time_t
fs_mktime       (struct  fs_tm         *p_time);
```

```
FS_DIR  *
fs_opendir      (const  char           *name_full);
```

```
int
fs_readdir      (FS_DIR                *p_dir,
                 struct fs_dirent      *p_dir_entry,
                 struct fs_dirent     **pp_result);
```

```
int
fs_remove       (const  char           *name_full);
```

```
int
fs_rename       (const  char           *name_full_old,
                 const  char           *name_full_new);
```

```
void
fs_rewind       (FS_FILE               *p_file);
```

```
int
fs_setbuf       (FS_FILE               *p_file,
                 fs_size_t              size);
```

```
int
fs_setvbuf      (FS_FILE               *p_file,
                 char                  *p_buf,
                 int                    mode,
                 fs_size_t              size);
```

fs_asctime_r()

```
char *fs_asctime_r (const  struct  fs_tm  *p_time,
                           char           *str_time);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Converts date/time to string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

Arguments

`p_time` Pointer to date/time to format.

`str_time` String buffer that will receive date/time string (see Note).

Returned Value

Pointer to `str_time`, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

`str_time` MUST be at least 26 characters long. Buffer overruns MUST be prevented by caller.

fs_chdir()

```
int fs_chdir (const  char  *path_dir);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_WORKING_DIR_EN

Set the working directory for the current task.

Arguments

`path_dir` String buffer that specifies EITHER...

(a) the absolute working directory path to set;

(b) a relative path that will be applied to the current working directory.

Returned Value

0, if no error occurs.

-1, otherwise

Notes/Warnings

None.

fs_clearerr()

```
void  fs_clearerr (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Clear EOF and error indicators on a file.

Arguments

`p_file` Pointer to a file.

Returned Value

None.

Notes/Warnings

None.

fs_closedir()

```
int  fs_closedir (FS_DIR  *p_dir);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG _DIR_EN

Close and free a directory.

Arguments

p_dir Pointer to a directory.

Returned Value

0, if the directory is successfully closed.
-1, if any error was encountered.

Notes/Warnings

After a directory is closed, the application MUST desist from accessing its directory pointer. This could cause file system corruption, since this handle may be re-used for a different directory.

fs_ctime_r()

```
char *fs_ctime_r (const  fs_time_t  *p_ts,
                         char       *str_time);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Converts timestamp to string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

Arguments

`p_ts` Pointer to timestamp to format.

`str_time` String buffer that will receive date/time string (see Note).

Returned Value

Pointer to `str_time`, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

`str_time` MUST be at least 26 characters long. Buffer overruns MUST be prevented by caller.

fs_fclose()

```
int  fs_fclose (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Close and free a file.

Arguments

p_file Pointer to a file.

Returned Value

0, if the file was successfully closed.

FS_EOF, otherwise.

Notes/Warnings

1. After a file is closed, the application MUST desist from accessing its file pointer. This could cause file system corruption, since this handle may be re-used for a different file.

2. If the most recent operation is output (write), all unwritten data is written to the file.

3. Any buffer assigned with fs_setbuf() or fs_setvbuf() shall no longer be accessed by the file system and may be re-used by the application.

fs_feof()

```
int  fs_feof (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Test EOF indicator on a file.

Arguments

p_file	Pointer to a file.

Returned Value

0,	if EOF indicator is NOT set or if an error occurred
Non-zero value,	if EOF indicator is set.

Notes/Warnings

1. The return value from this function should ALWAYS be tested against 0:

```
rtn = fs_feof(p_file);
if (rtn == 0) {
    // EOF indicator is NOT set
} else {
    // EOF indicator is     set
}
```

2. If the end-of-file indicator is set (i.e., `fs_feof()` returns `DEF_YES`), `fs_clearerr()` can be used to clear that indicator.

fs_ferror()

```
int  fs_ferror (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Test error indicator on a file.

Arguments

`p_file` Pointer to a file.

Returned Value

0, if error indicator is NOT set or if an error occurred

Non-zero value, if error indicator is set.

Notes/Warnings

1. The return value from this function should ALWAYS be tested against 0:

```
rtn = fs_ferror(p_file);
if (rtn == 0) {
    // Error indicator is NOT set
} else {
    // Error indicator is     set
}
```

2. If the error indicator is set (i.e., `fs_ferror()` returns a non-zero value), `fs_clearerr()` can be used to clear that indicator.

fs_fflush()

```
int  fs_fflush (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CF_FILE_BUF_EN

Flush buffer contents to file.

Arguments

p_file Pointer to a file.

Returned Value

0, if flushing succeeds.
FS_EOF, otherwise.

Notes/Warnings

1. If the most recent operation is output (write), all unwritten data is written to the file.

2. If the most recent operation is input (read), all buffered data is cleared.

fs_fgetpos()

```
int  fs_fgetpos (FS_FILE    *p_file,
                 fs_fpos_t  *p_pos);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Get file position indicator.

Arguments

`p_file` Pointer to a file.

`p_pos` Pointer to variable that will receive the file position indicator.

Returned Value

0, if no error occurs.
Non-zero value, otherwise.

Notes/Warnings

1. The return value should be tested against 0:

```
rtn = fs_fgetpos(p_file, &pos);
if (rtn == 0) {
    // No error occurred
} else {
    // Handle error
}
```

2. The value placed in `pos` should be passed to `FS_fsetpos()` to reposition the file to its position at the time when this function was called.

fs_flockfile()

```
void  fs_flockfile (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_FILE_LOCK_EN

Acquire task ownership of a file.

Arguments

p_file Pointer to a file.

Returned Value

None.

Notes/Warnings

A lock count is associated with each file:

1. The file is unlocked when the lock count is zero.
2. If the lock count is positive, a task owns the file.
3. When fs_flockfile() is called, if…
    a. …the lock count is zero OR
    b. …the lock count is positive and the caller owns the file…

    …the lock count will be incremented and the caller will own the file. Otherwise, the caller will wait until the lock count returns to zero.
4. Each call to fs_funlockfile() incremenets the lock count.
5. Matching calls to fs_flockfile() (or fs_ftrylockfile()) and fs_funlockfile() can be nested.

fs_fopen()

```
FS_FILE  *fs_fopen (const  char  *name_full,
                    const  char  *str_mode);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Open a file.

Arguments

`name_full` Name of the file. See Section 2.05 for information about file names.

`str_mode` Access mode of the file.

Returned Value

Pointer to a file, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

1. The access mode should be one of the strings shown in Table 4-2.
2. The character ‘b’ has no effect.
3. Opening a file with read mode fails if the file does not exist.
4. Opening a file with append mode causes all writes to be forced to the end-of-file.

fs_fread()

```
fs_size_t  fs_fread (void       *p_dest,
                     fs_size_t   size,
                     fs_size_t   nitems,
                     FS_FILE    *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Read from a file.

Arguments

p_dest	Pointer to destination buffer.
size	Size of each item to read.
nitems	Number of items to read.
p_file	Pointer to a file.

Returned Value

Number of items read.

Notes/Warnings

1. The size or nitems is 0, then the file is unchanged and zero is returned.

2. If the file is buffered and the last operation is output (write), then a call to fs_flush() or fs_fsetpos() or fs_fseek() MUST occur before input (read) can be performed.

3. The file must have been opened in read or update (read/write) mode.

fs_fseek()

```
int  fs_fseek (FS_FILE    *p_file,
               long  int   offset,
               int         origin);
```

File	Called from	Code enabled by
fs_api..c	Application; `fs_frewind()`	`FS_CFG_API_EN`

Set file position indicator.

Arguments

`p_file` Pointer to a file.

`offset` Offset from the file position specified by `whence`.

`origin` Reference position for offset:

`FS_SEEK_SET`	Offset is from the beginning of the file.
`FS_SEEK_CUR`	Offset is from the current file position.
`FS_SEEK_END`	Offset is from the end of the file.

Returned Value

0, if the function succeeds.
-1, otherwise.

Notes/Warnings

1. If a read or write error occurs, the error indicator shall be set.

2. The new file position, measured in bytes form the beginning of the file, is obtained by adding offset to…:

    a. …0 (the beginning of the file) if `whence` is `FS_SEEK_SET`;
    b. …the current file position if `whence` is `FS_SEEK_CUR`;
    c. …the file size if `whence` is `FS_SEEK_END`;

3. The end-of-file indicator is cleared.

4. If the file position indicator is set beyond the file's current data…

    a. …and data is later written to that point, reads from the gap will read 0.
    b. …the file MUST be opened in write or read/write mode.

fs_fsetpos()

```
int  fs_fsetpos (FS_FILE    *p_file,
                 fs_fpos_t  *p_pos);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Set file position indicator.

Arguments

`p_file` Pointer to a file.

`p_pos` Pointer to variable containing file position indicator.

Returned Value

0, if the function succeeds.
Non-zero value, otherwise.

Notes/Warnings

1. The return value should be tested against 0:

```
rtn = fs_fsetpos(pfile, &pos);
if (rtn == 0) {
    // No error occurred
} else {
    // Handle error
}
```

2. If a read or write error occurs, the error indicator shall be set.

3. The value stored in `pos` should be the value from an earlier call to `fs_fgetpos()`. No attempt is made to verify that the value in `pos` was obtained by a call to `fs_fgetpos()`.

4. See also `fs_fseek()`.

fs_ftell()

```
long  int  fs_ftell (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Get file position indicator.

Arguments

p_file	Pointer to a file.

Returned Value

The current file system position,	if the function succeeds.
-1,	otherwise.

Notes/Warnings

The file position returned is measured in bytes from the beginning of the file.

fs_ftruncate()

```
int  fs_ftruncate (FS_FILE   *p_file,
                   fs_off_t   size);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Truncate a file.

Arguments

p_file	Pointer to a file.
size	Size of the file after truncation

Returned Value

0,	if the function succeeds.
-1,	otherwise.

Notes/Warnings

1. The file MUST be opened in write or read/write mode.

2. If fs_ftruncate() succeeds, the size of the file shall be equal to length.

    a. If the size of the file was previously greater than length, the extra data shall no longer be available.
    b. If the file previously was smaller than this length, the size of the file shall be increased.

3. If the file position indicator before the call to fs_ftruncate() lay in the extra data destroyed by the function, then the file position will be set to the end-of-file.

fs_ftrylockfile()

```
int  fs_ftrylockfile (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_FILE_LOCK_EN

Acquire task ownership of a file (if available).

Arguments

p_file	Pointer to a file.

Returned Value

0,	if no error occurs and the file lock is acquired.
Non-zero value,	otherwise.

Notes/Warnings

fs_ftrylockfile() is the non-blocking version of fs_flockfile(); if the lock is not available, the function returns an error.

See fs_flockfile().

fs_funlockfile()

```
void  fs_funlockfile (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_FILE_LOCK_EN

Release task ownership of a file.

Arguments

p_file Pointer to a file.

Returned Value

None.

Notes/Warnings

See fs_flockfile().

fs_fwrite()

```
fs_size_t  fs_fwrite (void       *p_src,
                      fs_size_t   size,
                      fs_size_t   nitems,
                      FS_FILE    *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Write to a file.

Arguments

p_src	Pointer to source buffer.
size	Size of each item to write.
nitems	Number of items to write.
p_file	Pointer to a file.

Returned Value

Number of items written.

Notes/Warnings

1. The size or nitems is 0, then the file is unchanged and zero is returned.

2. If the file is buffered and the last operation is input (read), then a call to fs_fsetpos() or fs_fseek() MUST occur before output (write can be performed unless the end-of-file was encountered.

3. The file must have been opened in write or update (read/write) mode.

4. If the file was opened in append mode, all writes are forced to the end-of-file.

fs_getcwd()

```
char  *fs_getcwd (char       *path_dir,
                  fs_size_t   size)
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_WORKING_DIR_EN

Get the working directory for the current task.

Arguments

path_dir	String buffer that will receive the working directory path.
size	Size of string buffer.

Returned Value

Pointer to path_dir,	if no error occurs.
Pointer to NULL,	otherwise

Notes/Warnings

None.

fs_localtime_r()

```
struct  fs_tm  *fs_localtime_r (const  fs_time_t      *p_ts,
                                       struct  fs_tm  *p_time);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Convert timestamp to date/time.

Arguments

p_ts	Pointer to time value.
p_time	Pointer to variable that will receive broken-down time.

Returned Value

Pointer to p_time, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

None.

fs_mkdir()

```
int  fs_mkdir (const  char  *name_full);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Create a directory.

Arguments

`name_full` Name of the directory.

Returned Value

0, if the directory is created.
-1, if the directory is NOT created.

Notes/Warnings

None.

Example

Listing A-1. Using `fs_mkdir()`.

```
void  App_Fnct (void)
{
    int  err;

    .
    .
    .

    err = fs_mkdir("sd:0:\\data\\old");            /* Make dir.                                */
    if (err != 0) {
        APP_TRACE_INFO(("Could not make dir."));
    }

    .
    .
    .
}
```

fs_mktime()

```
fs_time_t  fs_mktime (struct  fs_tm  *p_time);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Convert date/time to timestamp.

Arguments

`p_time` Pointer to date/time to convert.

Returned Value

Time value, if NO errors.
`(fs_time_t)-1`, otherwise.

Notes/Warnings

None.

fs_opendir()

```
FS_DIR  *fs_opendir (const  char  *name_full);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_DIR_EN

Open a directory.

Arguments

name_full Name of the directory. See Section 2.05 for information about directory names.

Returned Value

Pointer to a directory, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

None.

fs_readdir_r()

```
int  fs_readdir (FS_DIR            *p_dir,
                 struct fs_dirent  *p_dir_entry,
                 struct fs_dirent **pp_result);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_DIR_EN

Read a directory entry from a directory.

Arguments

p_dir Pointer to a directory.

p_dir_entry Pointer to variable that will receive directory entry information.

pp_result Pointer to variable that will receive:

(a) p_dir_entry, if NO error occurs AND directory does not encounter EOF.

(b) pointer to NULL if an error occurs OR directory encounters EOF.

Returned Value

1, if an error occurs.
0, otherwise.

Notes/Warnings

1. Entries for “dot” (current directory) and “dot-dot” (parent directory) shall be returned, if present. No entry with an empty name shall be returned.

2. If an entry is removed from or added to the directory after the directory has been opened, information may or may not be returned for that entry.

fs_remove()

```
int  fs_remove (const  char  *name_full);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Delete a file or directory.

Arguments

`name_full` Name of the entry.

Returned Value

0, if the file is removed.
-1, if the file is NOT removed.

Notes/Warnings

1. When a file is removed, the space occupied by the file is freed and shall no longer be accessible.
2. A directory can be removed only if it is an empty directory.
3. The root directory cannot be removed.

Example

Listing A-2. Using `fs_remove()`.

```
void  App_Fnct (void)
{
    int  err;

    .
    .
    .

    err = fs_remove("sd:0:\\data\\file001.txt");   /* Remove file.                          */
    if (err != 0) {
        APP_TRACE_INFO(("Could not remove file."));
    }

    .
    .
    .

    err = fs_remove("sd:0:\\data\\old");           /* Remove dir.                           */
    if (err != 0) {
        APP_TRACE_INFO(("Could not remove dir."));
    }

    .
    .
    .
}
```

fs_rename()

```
int  fs_rename (const  char  *name_full_old,
                const  char  *name_full_new);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Rename a file or directory.

Arguments

name_full_old	Old name of the entry.
name_full_new	New name of the entry.

Returned Value

0, if the entry is renamed.
-1, if the entry is NOT renamed.

Notes/Warnings

1. name_full_old and name_full_new MUST specify entries on the same volume.
2. If path_old and path_new specify the same entry, the volume will not be modified and no error will be returned.
3. If path_old specifies a file:
    a. path_new must NOT specify a directory;
    b. if path_new is a file, it will be removed.
4. If path_old specifies a directory:
    a. path_new must NOT specify a file
    b. if path_new is a directory, path_new MUST be empty; if so, it will be removed.
5. The root directory may NOT be renamed.

Example

Listing A-3. Using `fs_rename()`.

```
void  App_Fnct (void)
{
    int  err;

    .
    .
    .

                                                /* See Note #1.                          */
    err = fs_rename("sd:0:\\data\\file001.txt",    /* Rename file.                       */
                    "sd:0:\\data\\old\\file001.txt");
    if (err != 0) {
        APP_TRACE_INFO(("Could not rename file."));
    }

    .
    .
    .
}
```

L4-6(1) For this example file rename to succeed, the following must be true when the function is called:

1. The file *sd:0:\data\file001.txt* must exist.
2. The directory *sd:0:\data\old* must exist.
3. If *sd:0:\data\old\file001.txt* exists, it must not be read-only.

If *sd:0:\data\old\file001.txt* exists and is not read-only, it will be removed and *sd:0:\data\file001.txt* will be renamed.

fs_rewind()

```
void  fs_rewind (FS_FILE  *p_file);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN

Reset file position indicator of a file.

Arguments

p_file	Pointer to a file.

Returned Value

None.

Notes/Warnings

1. `fs_rewind()` is equivalent to

   ```
   (void)fs_fseek(p_file, 0, FS_SEEK_SET)
   ```

   except that it also clears the error indictor of the file.

fs_rmdir()

```
int  fs_rmdir (const  char  *name_full);
```

File	Called from	Code enabled by
fs_api.c	Application	FS_CFG_API_EN and not FS_CFG_RD_ONLY_EN

Delete a directory.

Arguments

`name_full` Name of the file.

Returned Value

0, if the directory is removed.
-1, if the directory is NOT removed.

Notes/Warnings

1. A directory can be removed only if it is an empty directory.

2. The root directory cannot be removed.

Example

Listing A-4. Using `fs_rmdir()`.

```
void  App_Fnct (void)
{
    int  err;

    .
    .
    .

    err = fs_rmdir("sd:0:\\data\\old");            /* Remove dir.                          */
    if (err != 0) {
        APP_TRACE_INFO(("Could not remove dir."));
    }

    .
    .
    .
}
```

fs_setbuf()

```
int  fs_setbuf (FS_FILE    *p_file,
                fs_size_t   size);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_FILE_BUF_EN

Assign buffer to a file.

Arguments

p_file	Pointer to a file.
size	Size of buffer, in octets.

Returned Value

-1, if an error occurs.
0, if no error occurs.

Notes/Warnings

1. fs_setbuf() is equivalent to fs_setvbuf() invoked with FS__IOFBF for mode and FS_BUFSIZE for size.

fs_setvbuf()

```
int  fs_setvbuf (FS_FILE    *p_file,
                 char       *p_buf,
                 int         mode,
                 fs_size_t   size);
```

File	Called from	Code enabled by
fs_api..c	Application	FS_CFG_API_EN and FS_CFG_FILE_BUF_EN

Assign buffer to a file.

Arguments

p_file Pointer to a file.

p_buf Pointer to buffer.

mode Buffer mode:

FS__IONBR	Unbuffered.
FS__IOFBF	Fully buffered.

size Size of buffer, in octets.

Returned Value

-1, if an error occurs.
0, if no error occurs.

Notes/Warnings

1. fs_setvbuf() MUST be used after a stream is opened but before any other operation is performed on stream.

2. size MUST be more than or equal to the size of one sector; it will be rounded DOWN to the nearest size of a multiple of full sectors.

3. Once a buffer is assigned to a file, a new buffer may not be assigned nor may the assigned buffer be removed. To change the buffer, the file should be closed and re-opened.

4. Upon power loss, any data stored in file buffers will be lost.

A.03 DEVICE FUNCTIONS

Most device access functions can return any of the following device errors:

FS_ERR_DEV_INVALID_LOW_FMT	Device needs to be low-level formatted.
FS_ERR_DEV	Device access error.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NOT_OPEN	Device is not open.
FS_ERR_DEV_NOT_PRESENT	Device is not present.
FS_ERR_DEV_TIMEOUT	Device timeout error.

Each of these indicates that the state of the device is not suitable for the intended operation.

```
void
FSDev_Close                (CPU_CHAR              *name_dev,
                            FS_ERR                *p_err);
```

```
FS_PARTITION_NBR
FSDev_GetNbrPartitions     (CPU_CHAR              *name_dev,
                            FS_ERR                *p_err);
```

```
void
FSDev_GetDevName           (FS_QTY                 dev_nbr,
                            CPU_CHAR              *name_dev);
```

```
FS_QTY
FSDev_GetDevCnt            (void);
```

```
FS_QTY
FSDev_GetDevCntMax         (void);
```

```
void
FSDev_Open                 (CPU_CHAR              *name_dev,
                            void                  *p_dev_cfg,
                            FS_ERR                *p_err);
```

```
FS_PARTITION_NBR
FSDev_PartitionAdd         (CPU_CHAR              *name_dev,
                            FS_SEC_QTY             partition_size,
                            FS_ERR                *p_err);
```

```
void
FSDev_PartitionFind        (CPU_CHAR              *name_dev,
                            FS_PARTITION_NBR       partition_nbr,
                            FS_PARTITION_ENTRY    *p_partition_entry,
                            FS_ERR                *p_err);
```

```
void
FSDev_PartitionInit        (CPU_CHAR              *name_dev,
                            FS_SEC_QTY             partition_size,
                            FS_ERR                *p_err);
```

```
void
FSDev_Query                (CPU_CHAR              *name_dev,
                            FS_DEV_INFO           *p_info,
                            FS_ERR                *p_err);
```

```
void
FSDev_Rd            (CPU_CHAR           *name_dev,
                     void               *p_dest,
                     FS_SEC_NBR          start,
                     FS_SEC_QTY          cnt,
                     FS_ERR             *p_err);
```

```
CPU_BOOLEAN
FSDev_Refresh       (CPU_CHAR           *name_dev,
                     FS_ERR             *p_err);
```

```
void
FSDev_Wr            (CPU_CHAR           *name_dev,
                     void               *p_src,
                     FS_SEC_NBR          start,
                     FS_SEC_QTY          cnt,
                     FS_ERR             *p_err);
```

FSDev_Close()

```
void  FSDev_Close (CPU_CHAR  *name_dev,
                   FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Close and free a device.

Arguments

`name_dev` Device name.

`p_err` Pointer to variable that will receive return error code from this function :

`FS_ERR_NONE`	Device removed successfully.
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer

Returned Value

None.

Notes/Warnings

None.

FSDev_GetDevName()

```
void  FSDev_GetDevName (FS_QTY     dev_nbr,
                        CPU_CHAR  *name_dev);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Get name of the nth open device. `n` should be between 0 and the return value of `FSDev_GetNbrDevs()` (inclusive).

Arguments

`dev_nbr` Device number.

`name_dev` String buffer that will receive the device name (see Note #2).

Returned Value

None.

Notes/Warnings

1. `name_dev` MUST point to a character array of `FS_CFG_MAX_DEV_NAME_LEN` characters.
2. If the device does not exist, `name_dev` will receive an empty string.

FSDev_GetDevCnt()

```
FS_QTY  FSDev_GetDevCnt (void);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Gets the number of open devices.

Arguments

None.

Returned Value

Number of devices currently open.

Notes/Warnings

None.

FSDev_GetDevCntMax()

```
FS_QTY  FSDev_GetDevCntMax (void);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Gets the maximum possible number of open devices.

Arguments

None.

Returned Value

Maximum number of open devices.

Notes/Warnings

None.

FSDev_GetNbrPartitions()

```
FS_PARTITION_NBR  FSDev_GetNbrPartitions (CPU_CHAR  *name_dev,
                                          FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	FS_CFG_PARTITION_EN

Get number of partitions on a device

Arguments

`name_dev` Pointer to the device name.

`p_err` Pointer to variable that will receive return error code from this function.

FS_ERR_NONE	Number of partitions obtained.
FS_ERR_DEV_VOL_OPEN	Volume open on device.
FS_ERR_INVALID_SIG	Invalid MBR signature.
FS_ERR_NAME_NULL	Argument `name_dev` passed a NULL pointer.

Or device access error (see Section A.02).

Returned Value

Number of partitions on the device, if no error was encountered.
Zero, otherwise.

Notes/Warnings

Device state change will result from device I/O, not present or timeout error.

FSDev_Open()

```
void  FSDev_Open (CPU_CHAR  *name_dev,
                  void      *p_dev_cfg,
                  FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Open a device.

Arguments

`name_dev` Device name. See Section 2.04 for information about device names.

`p_dev_cfg` Pointer to device configuration.

`p_err` Pointer to variable that will receive the return error code from this function (see Note #2):

`FS_ERR_NONE`	Device opened successfully.
`FS_ERR_DEV_ALREADY_OPEN`	Device is already open.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_INVALID_NAME`	Specified device name not valid.
`FS_ERR_DEV_INVALID_SEC_SIZE`	Invalid device sector size.
`FS_ERR_DEV_INVALID_SIZE`	Invalid device size.
`FS_ERR_DEV_INVALID_UNIT_NBR`	Specified unit number invalid.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_NONE_AVAIL`	No devices available.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_TIMEOUT`	Device timeout error.
`FS_ERR_DEV_UNKNOWN`	Unknown device error.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer

Returned Value

None.

Notes/Warnings

1. The return error code from the function SHOULD always be checked by the calling application to determine whether the device was successfully opened. Repeated calls to `FSDev_Open()` resulting in errors that do not indicate failure to open (like `FS_ERR_DEV_LOW_FMT_INVALID`) without matching `FSDev_Close()` calls may exhaust the supply of device structures.

    a. If `FS_ERR_NONE` is returned, then the device has been added to the file system and is immediately accessible.
    b. If `FS_DEV_INVALID_LOW_FMT` is returned, then the device has been added to the file system, but needs to be low-level formatted, though it is present.

c. If FS_ERR_DEV_NOT_PRESENT, FS_ERR_DEV_IO or FS_ERR_DEV_TIMEOUT is returned, then the device has been added to the file system, though it is probably not present. The device will need to be either closed and re-added, or refreshed.
d. If FS_ERR_DEV_INVALID_NAME, FS_ERR_DEV_INVALID_SEC_SIZE, FS_ERR_DEV_INVALID_SIZE, FS_ERR_DEV_INVALID_UNIT_NBR or FS_ERR_DEV_NONE_AVAIL is returned, then the device has NOT been added to the file system.
e. If FS_ERR_DEV_UNKNOWN is returned, then the device driver is in an indeterminate state. The system MAY need to be restarted and the device driver should be examined for errors. The device has NOT been added to the file system.

FSDev_PartitionAdd()

```
FS_PARTITION_NBR  FSDev_PartitionAdd (CPU_CHAR    *name_dev,
                                      FS_SEC_QTY   partition_size,
                                      FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	`FS_CFG_PARTITION_EN` and not `FS_CFG_RD_ONLY_EN`

Adds a partition to a device.
See also Section 3.01.03.

Arguments

`name_dev` Device name

`partition_size` Size, in sectors, of the partition to add.

`p_err` Pointer to variable that will receive return error code from this function.

`FS_ERR_NONE`	Partition added.
`FS_ERR_INVALID_PARTITION`	Invalid partition.
`FS_ERR_INVALID_SEC_NBR`	Sector start or count invalid.
`FS_ERR_INVALID_SIG`	Invalid MBR signature.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.

Or device access error (see Section A.02).

Returned Value

The index of the created partition. The first partition on the device has an index of 0.
FS_INVALID_PARTITION_NBR is returned if the function fails to add the partition.

Notes/Warnings

Device state change will result from device I/O, not present or timeout error.

FSDev_PartitionFind()

```
void  FSDev_PartitionFind (CPU_CHAR            *name_dev,
                           FS_PARTITION_NBR     partition_nbr,
                           FS_PARTITION_ENTRY  *p_partition_entry,
                           FS_ERR              *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	FS_CFG_PARTITION_EN

Find a partition on a device.
See also Section 3.01.03.

Arguments

`name_dev` Device name.

`partition_nbr` Index of the partition to find.

`p_partition_entry` Pointer to variable that will receive the partition information.

`p_err` Pointer to variable that will receive return error code from this function.

FS_ERR_NONE	Partition found.
FS_ERR_DEV_VOL_OPEN	Volume open on device.
FS_ERR_INVALID_PARTITION	Invalid partition.
FS_ERR_INVALID_SEC_NBR	Sector start or count invalid.
FS_ERR_INVALID_SIG	Invalid MBR signature.
FS_ERR_NAME_NULL	Argument name_dev passed a NULL pointer.
FS_ERR_NULL_PTR	Argument p_partition_entry passed a NULL pointer.

Or device access error (see Section A.02).

Returned Value

None.

Notes/Warnings

Device state change will result from device I/O, not present or timeout error.

FSDev_PartitionInit()

```
void  FSDev_PartitionInit (CPU_CHAR    *name_dev,
                           FS_SEC_QTY   partition_size,
                           FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	not FS_CFG_RD_ONLY_EN

Initialize the partition structure on a device.
See also Section 3.01.03.

Arguments

`name_dev` Device name.

`partition_size` Size of partition, in sectors.
OR
0, if partition will occupy entire device.

`p_err` Pointer to variable that will receive the return error code from this function.

FS_ERR_NONE	Partition structure initialized.
FS_ERR_DEV_VOL_OPEN	Volume open on device.
FS_ERR_INVALID_SEC_NBR	Sector start or count invalid.
FS_ERR_NAME_NULL	Argument `name_dev` passed a NULL pointer.

Or device access error (see Section A.02).

Returned Value

None.

Notes/Warnings

1. Function blocked if a volume is open on the device. All volume (and files) MUST be closed prior to initializing the partition structure, since it will obliterate any existing file system.

2. Device state change will result from device I/O, not present or timeout error.

FSDev_Query()

```
void  FSDev_Query (CPU_CHAR     *name_dev,
                   FS_DEV_INFO  *p_info,
                   FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Obtain information about a device.

Arguments

`name_dev` Device name.

`p_info` Pointer to structure that will receive device information (see Note).

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Device information obtained.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_info` passed a NULL pointer.
`FS_ERR_INVALID_SEC_NBR`	Sector start or count invalid.

Or device access error (see Section A.02).

Returned Value

None.

Notes/Warnings

None.

FSDev_Rd()

```
void  FSDev_Rd (CPU_CHAR    *name_dev,
                void        *p_dest,
                FS_SEC_NBR   start,
                FS_SEC_QTY   cnt,
                FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Read data from device sector(s).

Arguments

`name_dev` Device name.

`p_dest` Pointer to destination buffer.

`start` Start sector of read.

`cnt` Number of sectors to read

`p_err` Pointer to variable that will receive the return error code from this function

`FS_ERR_NONE`	Sector(s) read.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_dest` passed a NULL pointer.

Or device access error (see Section A.02).

Returned Value

None.

Notes/Warnings

Device state change will result from device I/O, not present or timeout error.

FSDev_Refresh()

```
CPU_BOOLEAN  FSDev_Refresh (CPU_CHAR  *name_dev,
                            FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	N/A

Refresh a device.

Arguments

`name_dev` Device name.

`P_err` Pointer to variable that will receive the return error code from this function.

`FS_ERR_NONE`	Device opened successfully.
`FS_ERR_DEV_INVALID_SEC_SIZE`	Invalid device sector size.
`FS_ERR_DEV_INVALID_SIZE`	Invalid device size.
`FS_ERR_DEV_INVALID_UNIT_NBR`	Specified unit number invalid.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer

Or device access error (see Section A.02).

Returned Value

`DEF_YES`, if the device has changed.
`DEF_NO`, if the device has not changed.

Notes/Warnings

1. If device has changed, all volumes open on the device must be refreshed and all files closed and reopened.

2. A device status change may be caused by

    a. A device was connected, but no longer is.
    b. A device was not connected, but now is.
    c. A different device is connected.

FSDev_Wr()

```
void  FSDev_Wr (CPU_CHAR    *name_dev,
                void        *p_src,
                FS_SEC_NBR   start,
                FS_SEC_QTY   cnt,
                FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev.c	Application	not FS_CFG_RD_ONLY_EN

Write data to device sector(s).

Arguments

`name_dev` Device name.

`p_src` Pointer to source buffer.

`start` Start sector of write.

`cnt` Number of sectors to write

`p_err` Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Sector(s) written.
FS_ERR_NAME_NULL	Argument name_dev passed a NULL pointer.
FS_ERR_NULL_PTR	Argument p_src passed a NULL pointer.

Or device access error (see Section A.02).

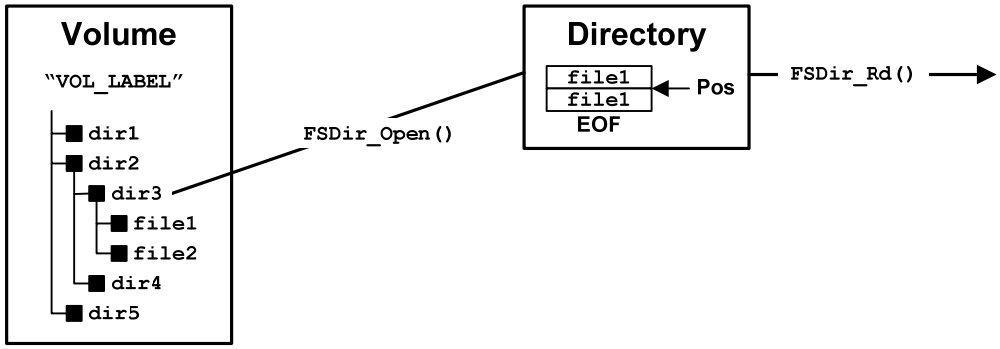
Returned Value

None.

Notes/Warnings

Device state change will result from device I/O, not present or timeout error.

A.04 DIRECTORY ACCESS FUNCTIONS




```
void
FSDir_Close (FS_DIR        *p_dir,
             FS_ERR        *p_err);
```

```
FS_DIR  *
FSDir_Open  (CPU_CHAR      *name_full,
             FS_ERR        *p_err);
```

```
void
FSDir_Rd    (FS_DIR        *p_dir,
             FS_DIR_ENTRY  *p_dir_entry,
             FS_ERR        *p_err);
```

FSDir_Close()

```
void  FSDir_Close (FS_DIR  *p_dir,
                   FS_ERR  *p_err);
```

File	Called from	Code enabled by
fs_dir.c	Application; `fs_closedir()`	`FS_CFG_DIR_EN`

Close and free a directory.

See `fs_closedir()` for more information.

Arguments

`p_dir` Pointer to a directory.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Directory closed.
`FS_ERR_NULL_PTR`	Argument `p_dir` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_dir`'s TYPE is invalid or unknown.
`FS_ERR_DIR_DIS`	Directory module disabled.
`FS_ERR_DIR_NOT_OPEN`	Directory NOT open.

Returned Value

None.

Notes/Warnings

None.

FSDir_Open()

```
FS_DIR  *FSDir_Open (CPU_CHAR  *name_full,
                     FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dir.c	Application; `fs_opendir()`	`FS_CFG_DIR_EN`

Open a directory.

See `fs_opendir()` for more information.

Arguments

`name_full` Name of the directory.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Directory opened.
`FS_ERR_NULL_PTR`	Argument `name_full` passed a NULL pointer.
`FS_ERR_DIR_DIS`	Directory module disabled.
`FS_ERR_DIR_NONE_AVAIL`	No directory available.
`FS_ERR_DEV`	Device access error.
`FS_ERR_NAME_INVALID`	Entry name specified invalid or volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name is too long.
`FS_ERR_VOL_NOT_OPEN`	Volume not opened.
`FS_ERR_VOL_NOT_MOUNTED`	Volume not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.

Or entry error (see Section B.04).

Returned Value

Pointer to a directory, if NO errors.
Pointer to NULL, otherwise.

Notes/Warnings

None.

FSDir_Rd()

```
void  FSDir_Rd (FS_DIR        *p_dir,
                FS_DIR_ENTRY  *p_dir_entry,
                FS_ERR        *p_err);
```

File	Called from	Code enabled by
fs_dir.c	Application; `fs_readdir_r()`	`FS_CFG_DIR_EN`

Read a directory entry from a directory.

See `fs_readdir_r()` for more information.

Arguments

`p_dir` Pointer to a directory.

`p_dir_entry` Pointer to variable that will receive directory entry information.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Directory read successfully.
`FS_ERR_NULL_PTR`	Argument `p_dir`/`p_dir_entry` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_dir`'s TYPE is invalid or unknown.
`FS_ERR_DIR_DIS`	Directory module disabled.
`FS_ERR_DIR_NOT_OPEN`	Directory NOT open.
`FS_ERR_EOF`	End of directory reached.
`FS_ERR_DEV`	Device access error.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.

Returned Value

None.

Notes/Warnings

None.

A.05 ENTRY ACCESS FUNCTIONS

```
void
FSEntry_AttribSet (CPU_CHAR       *name_full,
                   FS_FLAGS        attrib,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Copy      (CPU_CHAR       *name_full_src,
                   CPU_CHAR       *name_full_dest,
                   CPU_BOOLEAN     excl,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Create    (CPU_CHAR       *name_full,
                   CPU_BOOLEAN     dir,
                   CPU_BOOLEAN     excl,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Del       (CPU_CHAR       *name_full,
                   CPU_BOOLEAN     file,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Query     (CPU_CHAR       *name_full,
                   FS_ENTRY_INFO  *p_info,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Rename    (CPU_CHAR       *name_full_src,
                   CPU_CHAR       *name_full_dest,
                   CPU_BOOLEAN     excl,
                   FS_ERR         *p_err);
```

```
void
FSEntry_TimeSet   (CPU_CHAR       *name_full,
                   FS_DATE_TIME   *p_time,
                   CPU_INT08U      flag,
                   FS_ERR         *p_err);
```

FSEntry_AttribSet()

```
void  FSEntry_AttribSet (CPU_CHAR  *name_full,
                         FS_FLAGS   attrib,
                         FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application	not FS_CFG_RD_ONLY_EN

Set a file or directory's attributes.

Arguments

`name_full` Name of the entry.

`attrib` Entry attributes to set (see Note #2).

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Entry attributes set successfully.
FS_ERR_NULL_PTR	Argument `name_full` passed a NULL pointer.
FS_ERR_NAME_INVALID	Entry name specified invalid OR volume could not be found.
FS_ERR_NAME_PATH_TOO_LONG	Entry name specified too long.
FS_ERR_VOL_NOT_OPEN	Volume was not open.
FS_ERR_VOL_NOT_MOUNTED	Volume was not mounted.
FS_ERR_BUF_NONE_AVAIL	Buffer not available.
FS_ERR_DEV	Device access error.
Or entry error.	

Returned Value

None.

Notes/Warnings

1. If the entry does not exist, an error is returned.

2. Three attributes may be modified by this function:

FS_ENTRY_ATTRIB_RD	Entry is readable.
FS_ENTRY_ATTRIB_WR	Entry is writable.
FS_ENTRY_ATTRIB_HIDDEN	Entry is hidden from user-level processes.

   An attribute will be cleared if its flag is not OR'd into `attrib`. An attribute will be set if its flag is OR'd into `attrib`. If another flag besides these are set, then an error will be returned.

3. The attributes of the root directory may NOT be set.

FSEntry_Copy()

```
void  FSEntry_Copy (CPU_CHAR     *name_full_src,
                    CPU_CHAR     *name_full_dest,
                    CPU_BOOLEAN   excl,
                    FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application	not FS_CFG_RD_ONLY_EN

Copy a file.

Arguments

`name_full_src` Name of the source file.

`name_full_dest` Name of the destination file.

`excl` Indicates whether the creation of the new entry shall be exclusive (see Note #1) :
`DEF_YES`, if the entry shall be copied only if `name_full_dest` does not exist.
`DEF_NO`, if the entry shall be copied even if `name_full_dest` does exist.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File copied successfully.
`FS_ERR_NULL_PTR`	Argument `name_full_src` or `name_full_dest` passed a NULL pointer.
`FS_ERR_NAME_INVALID`	Entry name specified invalid OR volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name specified too long.
`FS_ERR_VOL_NOT_OPEN`	Volume was not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume was not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.
`FS_ERR_DEV`	Device access error.

Or entry error.

Returned Value

None.

Notes/Warnings

1. `name_full_src` must be an existing file. It may not be an existing directory.

2. If `excl` is `DEF_NO`, `name_full_dest` must either not exist or be an existing file; it may not be an existing directory. If `excl` is `DEF_YES`, `name_full_dest` must not exist.

FSEntry_Create()

```
void   FSEntry_Create (CPU_CHAR     *name_full,
                       CPU_BOOLEAN   dir,
                       CPU_BOOLEAN   excl,
                       FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application; `fs_mkdir()`	not `FS_CFG_RD_ONLY_EN`

Create a file or directory.

See also `fs_mkdir()`.

Arguments

`name_full` Name of the entry.

`dir` Indicates whether the new entry shall be a directory (see Note #1) :

- `DEF_YES`, if the entry shall be a directory.
- `DEF_NO`, if the entry shall be a file.

`excl` Indicates whether the creation of the new entry shall be exclusive (see Note #1) :

- `DEF_YES`, if the entry shall be created only if `p_name_full` does not exist.
- `DEF_NO`, if the entry shall be created even if `p_name_full` does exist.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Entry created successfully.
`FS_ERR_NULL_PTR`	Argument `name_full` passed a NULL pointer.
`FS_ERR_NAME_INVALID`	Entry name specified invalid OR volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name specified too long.
`FS_ERR_VOL_NOT_OPEN`	Volume was not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume was not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.
`FS_ERR_DEV`	Device access error.

Or entry error.

Returned Value

None.

Notes/Warnings

1. If the entry exists and is a file, `dir` is `DEF_NO` and `excl` is `DEF_NO`, then the existing entry will be truncated. If the entry exists and is a directory and `dir` is `DEF_YES`, then no change will be made to the file system.

2. If the entry exists and is a directory, `dir` is `DEF_NO` and `excl` is `DEF_NO`, then no change will be made to the file system. Similarly, if the entry exists and is a file, `dir` is `DEF_YES` and `excl` is `DEF_NO`, then no change will be made to the file system.

3. The root directory may not be created.

FSEntry_Del()

```
void  FSEntry_Del (CPU_CHAR     *name_full,
                   CPU_BOOLEAN   file,
                   FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application; `fs_rmdir()`; `fs_remove()`	not `FS_CFG_RD_ONLY_EN`

Delete a file or directory.

See also `fs_remove()` and `fs_rmdir()`.

Arguments

`name_full` Pointer to character string representing the name of the entry.

`file` Indicates whether the entry MAY be a file (see Notes #1 and #2):

- `DEF_YES`, if the entry may be a file.
- `DEF_NO`, if the entry may NOT be a file.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Entry date/time set successfully.
`FS_ERR_NULL_PTR`	Argument `name_full` passed a NULL pointer.
`FS_ERR_NAME_INVALID`	Entry name specified invalid OR volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name specified too long.
`FS_ERR_VOL_NOT_OPEN`	Volume was not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume was not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.
`FS_ERR_DEV`	Device access error.

Or entry error.

Returned Value

None.

Notes/Warnings

1. When a file is removed, the space occupied by the file is freed and shall no longer be accessible.

2. A directory can be removed only if it is an empty directory.

3. The root directory cannot be deleted.

FSEntry_Query()

```
void  FSEntry_Query (CPU_CHAR       *name_full,
                     FS_ENTRY_INFO  *p_info,
                     FS_ERR         *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application; `fs_stat()`	N/A

Get information about a file or directory.

Arguments

`name_full` Name of the entry.

`p_info` Pointer to structure that will receive the file information.

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File information obtained successfully.
`FS_ERR_NULL_PTR`	Argument `name_full` passed a NULL pointer.
`FS_ERR_NAME_INVALID`	Entry name specified invalid OR volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name specified too long.
`FS_ERR_VOL_NOT_OPEN`	Volume was not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume was not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.
`FS_ERR_DEV`	Device access error.

Returned Value

None.

Notes/Warnings

None.

FSEntry_Rename()

```
void  FSEntry_Rename (CPU_CHAR     *name_full_src,
                      CPU_CHAR     *name_full_dest,
                      CPU_BOOLEAN   excl,
                      FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application; `fs_rename()`	not `FS_CFG_RD_ONLY_EN`

Rename a file or directory.

See also `fs_rename()`.

Arguments

`name_full_old` Old path of the entry.

`name_full_new` New path of the entry.

`excl` Indicates whether the creation of the new entry shall be exclusive (see Note #1) :

`DEF_YES`, if the entry shall be renamed only if `name_full_new` does not exist.

`DEF_NO`, if the entry shall be renamed even if `name_full_new` does exist.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File copied successfully.
`FS_ERR_NULL_PTR`	Argument `name_full_old` or `name_full_new` passed a NULL pointer.
`FS_ERR_NAME_INVALID`	Entry name specified invalid OR volume could not be found.
`FS_ERR_NAME_PATH_TOO_LONG`	Entry name specified too long.
`FS_ERR_VOL_NOT_OPEN`	Volume was not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume was not mounted.
`FS_ERR_BUF_NONE_AVAIL`	Buffer not available.
`FS_ERR_DEV`	Device access error.
`FS_ERR_NAME_INVALID`	Invalid file name or path.

Or entry error.

Returned Value

None.

Notes/Warnings

1. If `name_full_old` and `name_full_new` specify entries on different volumes, then `name_full_old` MUST specify a file. If `name_full_old` specifies a directory, an error will be returned.

2. If `name_full_old` and `name_full_new` specify the same entry, the volume will not be modified and no error will be returned.

3. If `name_full_old` specifies a file:
   a. `name_full_new` must NOT specify a directory;
   b. if `excl` is `DEF_NO` and `name_full_new` is a file, it will be removed.

4. If `name_full_old` specifies a directory:
   a. `name_full_new` must NOT specify a file
   b. if `excl` is `DEF_NO` and `name_full_new` is a directory, `name_full_new` MUST be empty; if so, it will be removed.

5. If `excl` is `DEF_NO`, `name_full_new` must not exist.

6. The root directory may NOT be renamed.

FSEntry_TimeSet()

```
void  FSEntry_TimeSet (CPU_CHAR      *name_full,
                       FS_DATE_TIME  *p_time,
                       CPU_INT08U     flag,
                       FS_ERR        *p_err);
```

File	Called from	Code enabled by
fs_entry.c	Application	not FS_CFG_RD_ONLY_EN

Set a file or directory's date/time.

Arguments

`name_full` Name of the entry.

`p_time` Pointer to date/time.

`flag` Flag to indicate which Date/Time should be set

FS_DATE_TIME_CREATE	Entry Created Date/Time will be set.
FS_DATE_TIME_MODIFY	Entry Modified Date/Time will be set.
FS_DATE_TIME_ACCESS	Entry Accessed Date will be set.
FS_DATE_TIME_ALL	All the above will be set.

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Entry date/time set successfully.
FS_ERR_NULL_PTR	Argument name_full or p_time passed a NULL pointer.
FS_ERR_FILE_INVALID_DATE_TIME	Date/time specified invalid.
FS_ERR_NAME_INVALID	Entry name specified invalid OR volume could not be found.
FS_ERR_NAME_PATH_TOO_LONG	Entry name specified too long.
FS_ERR_VOL_NOT_OPEN	Volume was not open.
FS_ERR_VOL_NOT_MOUNTED	Volume was not mounted.
FS_ERR_BUF_NONE_AVAIL	Buffer not available.
FS_ERR_DEV	Device access error.

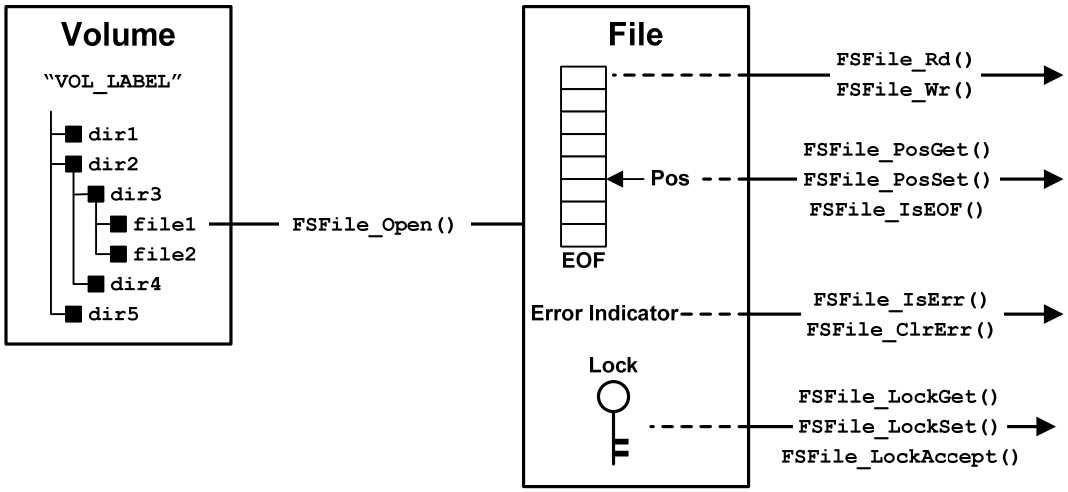
Returned Value

None.

Notes/Warnings

None.

A.06 FILE FUNCTIONS




```
void
FSFile_BufAssign (FS_FILE          *p_file,
                  void             *p_buf,
                  FS_FLAGS          mode,
                  CPU_SIZE_T        size,
                  FS_ERR           *p_err);
```

```
void
FSFile_BufFlush  (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_Close     (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_ClrErr    (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
CPU_BOOLEAN
FSFile_IsEOF     (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
CPU_BOOLEAN
FSFile_IsErr     (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_LockAccept(FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_LockGet   (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_LockSet   (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
FS_FILE  *
FSFile_Open      (CPU_CHAR         *name_full,
                  FS_FLAGS          mode
                  FS_ERR           *p_err);
```

```
FS_FILE_SIZE
FSFile_PosGet    (FS_FILE          *p_file,
                  FS_ERR           *p_err);
```

```
void
FSFile_PosSet    (FS_FILE          *p_file,
                  FS_FILE_OFFSET    offset,
                  FS_FLAGS          origin,
                  FS_ERR           *p_err);
```

```
void
FSFile_Query     (FS_FILE          *p_file,
                  FS_ENTRY_INFO    *p_info,
                  FS_ERR           *p_err);
```

```
CPU_SIZE_T
FSFile_Rd        (FS_FILE          *p_file,
                  void             *p_dest,
                  CPU_SIZE_T        size,
                  FS_ERR           *p_err);
```

```
void
FSFile_Truncate  (FS_FILE          *p_file,
                  FS_FILE_SIZE      size,
                  FS_ERR           *p_err);
```

```
CPU_SIZE_T
FSFile_Wr        (FS_FILE          *p_file,
                  void             *p_src,
                  CPU_SIZE_T        size,
                  FS_ERR           *p_err);
```

FSFile_BufAssign()

```
void  FSFile_BufAssign (FS_FILE     *p_file,
                        void        *p_buf,
                        FS_FLAGS     mode,
                        CPU_SIZE_T   size,
                        FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_setbuf()`; `fs_setvbuf()`	`FS_CFG_FILE_BUF_EN`

Assign buffer to a file.

See `fs_setvbuf()` for more information.

Arguments

`p_file` Pointer to a file.

`p_buf` Pointer to buffer.

`mode` Buffer mode:

`FS_FILE_BUF_MODE_RD`	Data buffered for reads.
`FS_FILE_BUF_MODE_WR`	Data buffered for writes.
`FS_FILE_BUF_MODE_RD_WR`	Data buffered for reads and writes..

`size` Size of buffer, in octets.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	File buffer assigned.
`FS_ERR_NULL_PTR`	Argument `p_file` or `p_buf` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_INVALID_BUF_MODE`	Invalid buffer mode.
`FS_ERR_FILE_INVALID_BUF_SIZE`	Invalid buffer size.
`FS_ERR_FILE_BUF_ALREADY_ASSIGNED`	Buffer already assigned.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_BufFlush()

```
void  FSFile_BufFlush (FS_FILE  *p_file,
                       FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fflush()`	`FS_CFG_FILE_BUF_EN`

Flush buffer contents to file.

See `fs_fflush()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	File buffer flushed successfully.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_Close()

```
void  FSFile_Close (FS_FILE  *p_file,
                    FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fclose()`	N/A

Close and free a file.

See `fs_fclose()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File closed.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_ClrErr()

```
void  FSFile_ClrErr (FS_FILE  *p_file,
                     FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_clearerr()`	N/A

Clear EOF and error indicators on a file.

See `fs_clearerr()` for more information

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Error and end-of-file indicators cleared.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_IsEOF()

```
CPU_BOOLEAN  FSFile_IsEOF (FS_FILE  *p_file,
                           FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_feof()`	N/A

Test EOF indicator on a file.

See `fs_feof()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	EOF indicator obtained.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

`DEF_NO` if EOF indicator is NOT set or if an error occurred
`DEF_YES` if EOF indicator is set.

Notes/Warnings

None.

FSFile_IsErr()

```
CPU_BOOLEAN  FSFile_IsErr (FS_FILE  *p_file,
                           FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_ferr()`	N/A

Test error indicator on a file.

See `fs_ferror()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Error indicator obtained.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

`DEF_NO` if error indicator is NOT set or if an error occurred
`DEF_YES` if error indicator is set.

Notes/Warnings

None.

FSFile_LockAccept()

```
void  FSFile_LockAccept (FS_FILE  *p_file,
                         FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_ftrylockfile()`	`FS_CFG_FILE_LOCK_EN`

Acquire task ownership of a file (if available).

See `fs_flockfile()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File lock acquired.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.
`FS_ERR_FILE_LOCKED`	File owned by another task.

Returned Value

None.

Notes/Warnings

None.

FSFile_LockGet()

```
void  FSFile_LockGet (FS_FILE  *p_file,
                      FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_flockfile()`	`FS_CFG_FILE_LOCK_EN`

Acquire task ownership of a file.

See `fs_flockfile()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File lock acquired.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_LockSet()

```
void  FSFile_LockSet (FS_FILE  *p_file,
                      FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_funlockfile()`	`FS_CFG_FILE_LOCK_EN`

Release task ownership of a file.

See `fs_funlockfile()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File lock acquired.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.
`FS_ERR_FILE_NOT_LOCKED`	File NOT locked or locked by different task.

Returned Value

None.

Notes/Warnings

None.

FSFile_Open()

```
FS_FILE  *FSFile_Open (CPU_CHAR  *name_full,
                       FS_FLAGS   mode
                       FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fopen()`	N/A

Open a file.

See `fs_fopen()` for more information.

Arguments

`name_full` Name of the file. See Section 2.05 for information about file names.

`mode` File access mode (see Notes #1 and #2).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	File opened.
`FS_ERR_NULL_PTR`	Argument `p_name_full` passed a NULL pointer.

Or entry error (see Section B.04).

Returned Value

None.

Notes/Warnings

1. The access mode should be the logical OR of one or more flags :

`FS_FILE_ACCESS_MODE_RD`	File opened for reads.
`FS_FILE_ACCESS_MODE_WR`	File opened for writes.
`FS_FILE_ACCESS_MODE_CREATE`	File will be created, if necessary.
`FS_FILE_ACCESS_MODE_TRUNC`	File length will be truncated to 0.
`FS_FILE_ACCESS_MODE_APPEND`	All writes will be performed at EOF.
`FS_FILE_ACCESS_MODE_EXCL`	File will be opened if and only if it does not already exist.
`FS_FILE_ACCESS_MODE_CACHED`	File data will be cached.

   a. If `FS_FILE_ACCESS_MODE_TRUNC` is set, then `FS_FILE_ACCESS_MODE_WR` must also be set.
   b. If `FS_FILE_ACCESS_MODE_EXCL` is set, then `FS_FILE_ACCESS_MODE_CREATE` must also be set.
   c. `FS_FILE_ACCESS_MODE_RD` and/or `FS_FILE_ACCESS_MODE_WR` must be set.

2. The mode string argument of `fs_fopen()` function can specify a subset of the possible valid `mode`s for this function. The equivalent modes of `fs_fopen()` mode strings are shown in Table 5-4.

fopen() Mode String	mode Equivalent
"r" or "rb"	FS_FILE_ACCESS_MODE_RD
"w" or "wb"	FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE | FS_FILE_ACCESS_MODE_TRUNC
"a" or "ab"	FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE | FS_FILE_ACCESS_MODE_APPEND
"r+" or "rb+" or "r+b"	FS_FILE_ACCESS_MODE_RD | FS_FILE_ACCESS_MODE_WR
"w+" or "wb+" or "w+b"	FS_FILE_ACCESS_MODE_RD | FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE | FS_FILE_ACCESS_MODE_TRUNC
"a+" or "ab+" or "a+b"	FS_FILE_ACCESS_MODE_RD | FS_FILE_ACCESS_MODE_WR | FS_FILE_ACCESS_MODE_CREATE | FS_FILE_ACCESS_MODE_APPEND

Table A-2. fs_fopen() mode strings and mode equivalents.

FSFile_PosGet()

```
FS_FILE_SIZE  FSFile_PosGet (FS_FILE  *p_file,
                             FS_ERR   *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_ftell()`; `fs_fgetpos()`	N/A

Set file position indicator.

See `fs_ftell()` for more information.

Arguments

`p_file` Pointer to a file.

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File position gotten successfully.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.
`FS_ERR_FILE_INVALID_POS`	Invalid file position.

Returned Value

The current file position,	if no errors (see Note).
0,	otherwise.

Notes/Warnings

The file position returned is the number of bytes from the beginning of the file up to the current file position.

FSFile_PosSet()

```
void  FSFile_PosSet (FS_FILE         *p_file,
                     FS_FILE_OFFSET   offset,
                     FS_FLAGS         origin,
                     FS_ERR          *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fseek()`; `fs_fsetpos()`	N/A

Get file position indicator.

See `fs_fseek()` for more information.

Arguments

`p_file` Pointer to a file.

`offset` Offset from the file position specified by `origin`.

`origin` Reference position for offset:

`FS_FILE_ORIGIN_START`	Offset is from the beginning of the file.
`FS_FILE_ORIGIN_CUR`	Offset is from the current file position.
`FS_FILE_ORIGIN_END`	Offset is from the end of the file.

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File position set successfully.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_INVALID_ORIGIN`	Invalid origin specified.
`FS_ERR_FILE_INVALID_OFFSET`	Invalid offset specified.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_Query()

```
void  FSFile_Query (FS_FILE        *p_file,
                    FS_ENTRY_INFO  *p_info,
                    FS_ERR         *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fstat()`	N/A

`FSFile_Query()` is used to get information about a file.

Arguments

`p_file` Pointer to a file.

`p_info` Pointer to structure that will receive the file information (see Note).

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File information obtained successfully.
`FS_ERR_NULL_PTR`	Argument `p_file` or `p_info` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_Rd()

```
CPU_SIZE_T  FSFile_Rd (FS_FILE     *p_file,
                       void        *p_dest,
                       CPU_SIZE_T   size,
                       FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fread()`	N/A

Read from a file.

See `fs_fread()` for more information.

Arguments

`p_file` Pointer to a file.

`p_dest` Pointer to destination buffer.

`size` Number of octets to read.

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File read successfully.
`FS_ERR_EOF`	End-of-file reached.
`FS_ERR_NULL_PTR`	Argument `p_file`/`p_dest` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.
`FS_ERR_FILE_INVALID_OP`	Invalid operation on file.
`FS_ERR_DEV`	Device access error.

Returned Value

The number of bytes read, if file read successful.
0, otherwise.

Notes/Warnings

None.

FSFile_Truncate()

```
void  FSFile_Truncate (FS_FILE       *p_file,
                       FS_FILE_SIZE   size,
                       FS_ERR        *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_ftruncate()`	not `FS_CFG_RD_ONLY_EN`

Truncate a file.

See `fs_ftruncate()` for more information.

Arguments

`p_file` Pointer to a file.

`size` Size of the file after truncation

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File truncated successfully.
`FS_ERR_NULL_PTR`	Argument `p_file` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.

Returned Value

None.

Notes/Warnings

None.

FSFile_Wr()

```
CPU_SIZE_T  FSFile_Wr (FS_FILE     *p_file,
                       void        *p_src,
                       CPU_SIZE_T   size,
                       FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; `fs_fwrite()`	not `FS_CFG_RD_ONLY_EN`

Write to a file.

See `fs_fwrite()` for more information.

Arguments

`p_file` Pointer to a file.

`p_src` Pointer to source buffer.

`size` Number of octets to write.

`p_err` Pointer to variable that will the receive return error code from the function:

`FS_ERR_NONE`	File write successfully.
`FS_ERR_NULL_PTR`	Argument `p_file`/`p_src` passed a NULL pointer.
`FS_ERR_INVALID_TYPE`	Argument `p_file`'s type is invalid or unknown.
`FS_ERR_FILE_NOT_OPEN`	File NOT open.
`FS_ERR_FILE_INVALID_OP`	Invalid operation on file.
`FS_ERR_DEV`	Device access error.

Returned Value

The number of bytes written,	if file write successful.
0,	otherwise.

Notes/Warnings

None.

A.07 TIME FUNCTIONS

```
void
FSTime_TimeGet      (FS_DATE_TIME  *p_time,
                     FS_ERR        *p_err);
```

```
void
FSTime_TimeSet      (FS_DATE_TIME  *p_time,
                     FS_ERR        *p_err);
```

```
void
FSTime_Time_to_Str  (FS_DATE_TIME  *p_time,
                     CPU_CHAR      *str_time,
                     CPU_SIZE_T     len,
                     FS_ERR        *p_err);
```

```
void
FSTime_Time_to_TS   (FS_DATE_TIME  *p_time,
                     FS_TS         *p_ts,
                     FS_ERR        *p_err);
```

```
void
FSTime_Time_to_Str  (FS_TS          ts,
                     CPU_CHAR      *str_time,
                     CPU_SIZE_T     len,
                     FS_ERR        *p_err);
```

```
void
FSTime_TS_to_Time   (FS_TS          ts,
                     FS_DATE_TIME  *p_time,
                     FS_ERR        *p_err);
```

FSTime_TimeGet()

```
void  FSTime_TimeGet (FS_DATE_TIME  *p_time,
                      FS_ERR        *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application	N/A

Get current date/time.

Arguments

`p_time` Pointer to variable that will receive the current date/time.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Date/time gotten.
`FS_ERR_NULL_PTR`	Argument `p_time` passed a NULL pointer.
`FS_ERR_INVALID_TIME`	Invalid date/time.

Returned Value

None.

Notes/Warnings

None.

FSTime_TimeSet()

```
void  FSTime_TimeSet (FS_DATE_TIME  *p_time,
                      FS_ERR        *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application	N/A

Get current date/time.

Arguments

`p_time` Pointer to date/time to set.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Date/time set.
`FS_ERR_NULL_PTR`	Argument `p_time` passed a NULL pointer.
`FS_ERR_INVALID_TIME`	Invalid date/time.

Returned Value

None.

Notes/Warnings

None.

FSTime_Time_to_Str()

```
void  FSTime_Time_to_Str (FS_DATE_TIME  *p_time,
                          CPU_CHAR      *str_time,
                          CPU_SIZE_T     len,
                          FS_ERR        *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application; `fs_asctime_r()`; `FSTime_TS_to_Str()`	N/A

Convert date/time to string.

Arguments

`p_time` Pointer to the date/time to format.

`str_time` String buffer that will receive the date/time string.

`len` Length of string buffer.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	String formatted.
`FS_ERR_NULL_PTR`	Argument `p_time` or `str_time` passed a NULL pointer.
`FS_ERR_INVALID_LEN`	Invalid length.
`FS_ERR_INVALID_TIME`	Invalid date/time.

Returned Value

None.

Notes/Warnings

1. `str_time` MUST be at least 26 characters long. Buffer overruns MUST be prevented by caller.

2. The date/time is formatted into a string in the form:
   ```
   Sun Sep 16 01:03:52 1983\n\0
   ```

FSTime_Time_to_TS()

```
void  FSTime_Time_to_TS (FS_DATE_TIME  *p_time,
                         FS_TS         *p_ts,
                         FS_ERR        *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application; `fs_mktime()`	N/A

Convert date/time to timestamp.

Arguments

`p_time` Pointer to the date/time to format.

`p_ts` Pointer to variable that will receive the timestamp.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	String formatted.
`FS_ERR_NULL_PTR`	Argument `p_time` or `p_ts` passed a NULL pointer.
`FS_ERR_INVALID_TIME`	Invalid date/time.

Returned Value

None.

Notes/Warnings

1. The timestamp value is the number of seconds since January 1, 1970. No earlier date has a valid timestamp representation..
2. The timestamp can encode any date from January 1, 1970, until 2^{32} seconds after that date, slightly more than 136 years later (February 7, 2106).
3. The time stamp is computed using the `p_time` members `Year`, `Month`, `Day`, `Hour`, `Minute` and `Second`.
    a. Range restrictions of `Day`, `Hour`, `Minute` and `Second` are NOT enforced on the input. The month MUST be valid because the day of year must be determinable.

FSTime_TS_to_Str()

```
void  FSTime_Time_to_Str (FS_TS        ts,
                          CPU_CHAR    *str_time,
                          CPU_SIZE_T   len,
                          FS_ERR      *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application; `fs_ctime_r()`	N/A

Convert timestamp to string.

Arguments

`ts` Timestamp to format.

`str_time` String buffer that will receive the date/time string.

`len` Length of string buffer.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	String formatted.
`FS_ERR_NULL_PTR`	Argument `str_time` passed a NULL pointer.
`FS_ERR_INVALID_LEN`	Invalid length.
`FS_ERR_INVALID_TIME`	Invalid date/time.

Returned Value

None.

Notes/Warnings

1. `str_time` MUST be at least 26 characters long. Buffer overruns MUST be prevented by caller.

2. The date/time is formatted into a string in the form:
   ```
   Sun Sep 16 01:03:52 1983\n\0
   ```

FSTime_TS_to_Time()

```
void  FSTime_TS_to_Time (FS_TS          ts,
                         FS_DATE_TIME  *p_time,
                         FS_ERR        *p_err)
```

File	Called from	Code enabled by
fs_time.c	Application; `fs_localtime_r()`; `FSTime_TS_to_Str()`	N/A

Convert timestamp to date/time.

Arguments

`ts` Timestamp to convert.

`p_str` Pointer to variable that will receive the date/time.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Timestamp converted.
`FS_ERR_NULL_PTR`	Argument `p_time` passed a NULL pointer.
`FS_ERR_INVALID_TIMESTAMP`	Invalid timestamp.

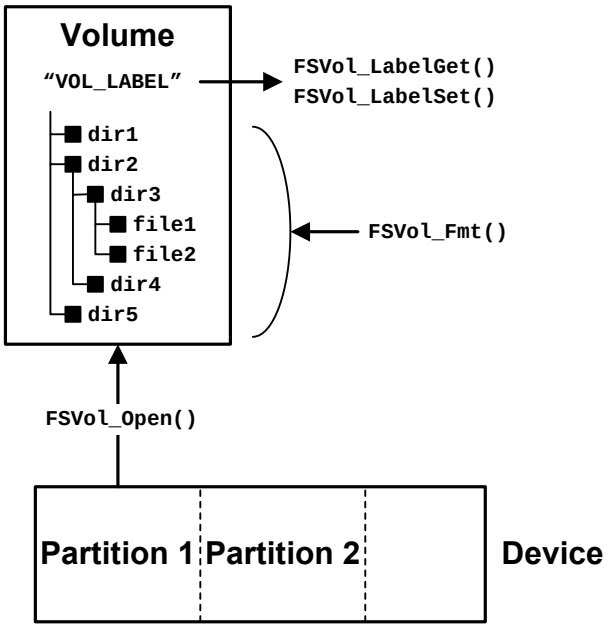
Returned Value

None.

Notes/Warnings

ALL `p_time` members, including `DayOfWeek` and `DayOfYear` are assigned correct values.

A.08 VOLUME FUNCTIONS




```
void
FSVol_Close             (CPU_CHAR           *name_vol,
                         FS_ERR             *p_err);
```

```
void
FSVol_Fmt               (CPU_CHAR           *name_vol,
                         void               *p_fs_cfg,
                         FS_ERR             *p_err);
```

```
void
FSVol_GetDfltVolName    (CPU_CHAR           *name_vol);
```

```
FS_QTY
FSVol_GetVolCnt         (void);
```

```
FS_QTY
FSVol_GetVolCntMax      (void);
```

```
void
FSVol_GetVolName        (FS_QTY              vol_nbr,
                         CPU_CHAR           *name_vol);
```

```
CPU_BOOLEAN
FSVol_IsMounted         (CPU_CHAR           *name_vol);
```

```
void
FSVol_LabelGet          (CPU_CHAR           *name_vol,
                         CPU_CHAR           *label,
                         CPU_SIZE_T          len_max,
                         FS_ERR             *p_err);
```

```
void
FSVol_LabelSet        (CPU_CHAR           *name_vol,
                       CPU_CHAR           *label,
                       FS_ERR             *p_err);
```

```
void
FSVol_Open            (CPU_CHAR           *name_vol,
                       CPU_CHAR           *name_dev,
                       FS_PARTITION_NBR    partition_nbr,
                       FS_ERR             *p_err);
```

```
void
FSVol_Query           (CPU_CHAR           *name_vol,
                       FS_VOL_INFO        *p_info,
                       FS_ERR             *p_err);
```

```
void
FSVol_Rd              (CPU_CHAR           *name_vol,
                       void               *p_dest,
                       FS_SEC_NBR          start,
                       FS_SEC_QTY          cnt,
                       FS_ERR             *p_err);
```

```
void
FSVol_Wr              (CPU_CHAR           *name_vol,
                       void               *p_src,
                       FS_SEC_NBR          start,
                       FS_SEC_QTY          cnt,
                       FS_ERR             *p_err);
```

FSVol_Close()

```
void  FSVol_Close (CPU_CHAR  *name_vol,
                   FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Close and free a volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will receive the return error code from this function. See Note #2.

`FS_ERR_NONE`	Volume opened.
`FS_ERR_NAME_NULL`	Argument `name_vol` passed a NULL pointer.
`FS_ERR_VOL_NOT_OPEN`	Volume not open.

Returned Value

None.

Notes/Warnings

None.

FSVol_Fmt()

```
void  FSVol_Fmt (CPU_CHAR  *name_vol,
                 void      *p_fs_cfg,
                 FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	not FS_CFG_RD_ONLY_EN

Format a volume.

Arguments

`name_vol` Colume name.

`p_fs_cfg` Pointer to file system driver-specific configuration. For all file system drivers, if this is a pointer to NULL, then the default configuration will be selected. More information about the appropriate structure for the FAT file system driver can be found in Chapter 6.

`p_err` Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Volume formatted.
FS_ERR_DEV	Device error.
FS_ERR_DEV_INVALID_SIZE	Invalid device size.
FS_ERR_NAME_NULL	Argument name_vol passed a NULL pointer.
FS_ERR_VOL_DIRS_OPEN	Directories open on volume.
FS_ERR_VOL_FILES_OPEN	Files open on volume.
FS_ERR_VOL_INVALID_SYS	Invalid file system parameters.
FS_ERR_VOL_NOT_OPEN	Volume not open.

Required Configuration

None.

Notes/Warnings

1. Function blocked if files or directories are open on the volume. All files and directories MUST be closed prior to formatting the volume.

2. For any file system driver, if `p_fs_cfg` is a pointer to NULL, then the default configuration will be selected. If non-NULL, the argument should be passed a pointer to the appropriate configuration structure. For the FAT file system driver, `p_fs_cfg` should be passed a pointer to a `FS_FAT_SYS_CFG`.

FSVol_GetDfltVolName()

```
void  FSVol_GetDfltVolName (CPU_CHAR  *name_vol);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Get name of the default volume.

Arguments

`name_vol` String buffer that will receive the volume name (see Note #2).

Returned Value

None.

Notes/Warnings

1. `name_vol` MUST point to a character array of `FS_CFG_MAX_VOL_NAME_LEN` characters.
2. If the volume does not exist, `name_vol` will receive an empty string.

FSVol_GetVolCnt()

```
FS_QTY  FSVol_GetVolCnt (void);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Get the number of open volumes.

Arguments

None.

Returned Value

Number of volumes currently open.

Notes/Warnings

None.

FSVol_GetVolCntMax()

```
FS_QTY  FSVol_GetVolCntMax (void);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Get the maximum possible number of open volumes.

Arguments

None.

Returned Value

The maximum number of open volumes.

Notes/Warnings

None.

FSVol_GetVolName()

```
void  FSVol_GetVolName (FS_QTY     vol_nbr,
                        CPU_CHAR  *name_vol);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Get name of the nth open volume. `n` should be between 0 and the return value of `FSVol_GetNbrVols()` (inclusive).

Arguments

`vol_nbr` Volume number.

`name_vol` String buffer that will receive the volume name (see Note #2).

Returned Value

None.

Notes/Warnings

1. `name_vol` MUST point to a character array of `FS_CFG_MAX_VOL_NAME_LEN` characters.
2. If the volume does not exist, `name_vol` will receive an empty string.

FSVol_IsDflt()

```
CPU_BOOLEAN  FSVol_IsDflt (CPU_CHAR  *name_vol);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Determine whether a volume is the default volume.

Arguments

`name_vol` Volume name.

Returned Value

`DEF_YES`, if the volume with name `name_vol` is the default volume.

`DEF_NO`, if no volume with name `name_vol` exists.
or the volume with name `name_vol` is not the default volume.

Notes/Warnings

None.

FSVol_IsMounted()

```
CPU_BOOLEAN  FSVol_IsMounted (CPU_CHAR  *name_vol);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Determine whether a volume is mounted.

Arguments

`name_vol` Volume name.

Returned Value

`DEF_YES` if the volume is open and is mounted.
`DEF_NO` if the volume is not open or is not mounted.

Notes/Warnings

None.

FSVol_LabelGet()

```
void  FSVol_LabelGet (CPU_CHAR    *name_vol,
                      CPU_CHAR    *label,
                      CPU_SIZE_T   len_max,
                      FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Get volume label.

Arguments

`name_vol` Volume name.

`label` String buffer that will receive volume label.

`len_max` Size of string buffer.

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Label gotten.
`FS_ERR_DEV_CHNGD`	Device has changed.
`FS_ERR_NAME_NULL`	Argument `name_vol` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `label` passed a NULL pointer.
`FS_ERR_DEV`	Device access error.
`FS_ERR_VOL_LABEL_NOT_FOUND`	Volume label was not found.
`FS_ERR_VOL_LABEL_TOO_LONG`	Volume label is too long.
`FS_ERR_VOL_NOT_MOUNTED`	Volume is not mounted.
`FS_ERR_VOL_NOT_OPEN`	Volume is not open.

Required Configuration

None.

Notes/Warnings

`len_max` is the maximum length string that can be stored in the buffer `label`; it does NOT include the final NULL character. The buffer label MUST be of at least `len_max + 1` characters..

FSVol_LabelSet()

```
void  FSVol_LabelSet (CPU_CHAR  *name_vol,
                      CPU_CHAR  *label,
                      FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	not FS_CFG_RD_ONLY_EN

Set volume label.

Arguments

`name_vol` Volume name.

`label` Volume label.

`p_err` Pointer to variable that will receive the return error code from this function:

FS_ERR_NONE	Label set.
FS_ERR_DEV_CHNGD	Device has changed.
FS_ERR_NAME_NULL	Argument `name_vol` passed a NULL pointer.
FS_ERR_NULL_PTR	Argument `label` passed a NULL pointer.
FS_ERR_DEV	Device access error.
FS_ERR_DIR_FULL	Directory is full (space could not be allocated).
FS_ERR_DEV_FULL	Device is full (space could not be allocated).
FS_ERR_VOL_LABEL_INVALID	Volume label is invalid.
FS_ERR_VOL_LABEL_TOO_LONG	Volume label is too long.
FS_ERR_VOL_NOT_MOUNTED	Volume is not mounted.
FS_ERR_VOL_NOT_OPEN	Volume is not open.

Returned Value

None.

Notes/Warnings

The label on a FAT volume must be no longer than 11-characters, each belonging to the set of valid short file name (SFN) characters. Before it is committed to the volume, the label will be converted to upper case and will be padded with spaces until it is an 11-character string.

FSVol_Open()

```
void  FSVol_Open (CPU_CHAR           *name_vol,
                  CPU_CHAR           *name_dev,
                  FS_PARTITION_NBR    partition_nbr,
                  FS_ERR             *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Open a volume.

Arguments

`name_vol` Volume name. See Section 2.04 for information about device names.

`name_dev` Device name.

`partition_nbr` Partition number. If 0, the default partition will be mounted.

`p_err` Pointer to variable that will receive the return error code from this function. See Note #2.

`FS_ERR_NONE`	Volume opened.
`FS_ERR_DEV_VOL_OPEN`	Volume open on device.
`FS_ERR_INVALID_SIG`	Invalid MBR signature.
`FS_ERR_NAME_NULL`	Argument `name_vol` / `name_dev` passed a NULL pointer.
`FS_ERR_PARTITION_INVALID_NBR`	Invalid partition number.
`FS_ERR_PARTITION_NOT_FOUND`	Partition not found.
`FS_ERR_VOL_ALREADY_OPEN`	Volume is already open.
`FS_ERR_VOL_INVALID_NAME`	Volume name invalid.
`FS_ERR_VOL_NONE_AVAIL`	No volumes available.

Or device access error (see Section A.02).

Returned Value

None.

Notes/Warnings

1. If `FS_ERR_PARTITION_NOT_FOUND` is returned, then no valid partition (or valid file system) was found on the device. It is still placed on the list of used volumes; however, it cannot be addressed as a mounted volume (e.g., files cannot be accessed). Thereafter, unless a new device is inserted, the only valid commands are

    a. `FSVol_Fmt()`, which creates a file system on the device;
    b. `FSVol_Close()`, which frees the volume structure;

c. `FSVol_Query()`, which returns information about the device.

2. If `FS_ERR_DEV`, `FS_ERR_DEV_NOT_PRESENT`, `FS_ERR_DEV_IO` or `FS_ERR_DEV_TIMEOUT` is returned, then the volume has been added to the file system, though the underlying device is probably not present. The volume will need to be either closed and re-added, or refreshed.

FSVol_Query()

```
void  FSVol_Query (CPU_CHAR     *name_vol,
                   FS_VOL_INFO  *p_info,
                   FS_ERR       *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Obtain information about a volume.

Arguments

`name_vol` Volume name.

`p_info` Pointer to structure that will receive volume information (see Note).

`p_err` Pointer to variable that will receive the return error code from this function:

`FS_ERR_NONE`	Volume information obtained.
`FS_ERR_DEV`	Device access error.
`FS_ERR_NAME_NULL`	Argument `name_vol` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_info` passed a NULL pointer.
`FS_ERR_VOL_NOT_OPEN`	Volume is not open.

Returned Value

None.

Notes/Warnings

None.

FSVol_Rd()

```
void  FSVol_Rd (CPU_CHAR    *name_vol,
                void        *p_dest,
                FS_SEC_NBR   start,
                FS_SEC_QTY   cnt,
                FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	N/A

Reads data from volume sector(s).

Arguments

`name_vol` Volume name.

`p_dest` Pointer to destination buffer.

`start` Start sector of read.

`cnt` Number of sectors to read

`p_err` Pointer to variable that will receive the return error code from this function

`FS_ERR_NONE`	Sector(s) read.
`FS_ERR_DEV`	Device access error.
`FS_ERR_NAME_NULL`	Argument `name_vol` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_dest` passed a NULL pointer.
`FS_ERR_VOL_NOT_MOUNTED`	Volume is not mounted.
`FS_ERR_VOL_NOT_OPEN`	Volume is not open.

Returned Value

None.

Required Configuration

None.

Notes/Warnings

None.

FSVol_Wr()

```
void  FSVol_Wr (CPU_CHAR    *name_vol,
                void        *p_src,
                FS_SEC_NBR   start,
                FS_SEC_QTY   cnt,
                FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_vol.c	Application	not FS_CFG_RD_ONLY_EN

Writes data to volume sector(s).

Arguments

`name_vol` Volume name.

`p_src` Pointer to source buffer.

`start` Start sector of write.

`cnt` Number of sectors to write

`p_err` Pointer to variable that will receive the return error code from this function

FS_ERR_NONE	Sector(s) written.
FS_ERR_DEV	Device access error.
FS_ERR_NAME_NULL	Argument `name_vol` passed a NULL pointer.
FS_ERR_NULL_PTR	Argument `p_src` passed a NULL pointer.
FS_ERR_VOL_NOT_MOUNTED	Volume is not mounted.
FS_ERR_VOL_NOT_OPEN	Volume is not open.

Returned Value

None.

Notes/Warnings

None.

A.09 VOLUME CACHE FUNCTIONS

```
void
FSVol_CacheAssign      (CPU_CHAR           *name_vol,
                        FS_VOL_CACHE_API   *p_cache_api,
                        void               *p_cache_data,
                        CPU_INT32U          size,
                        CPU_INT08U          pct_mgmt,
                        CPU_INT08U          pct_dir,
                        FS_FLAGS            mode,
                        FS_ERR             *p_err);
```

```
void
FSVol_CacheInvalidate (CPU_CHAR  *name_vol,
                       FS_ERR    *p_err);
```

```
void
FSVol_CacheFlush       (CPU_CHAR  *name_vol,
                        FS_ERR    *p_err);
```

FSVol_CacheAssign ()

```
void  FSVol_CacheAssign (CPU_CHAR           *name_vol,
                         FS_VOL_CACHE_API   *p_cache_api,
                         void               *p_cache_data,
                         CPU_INT32U          size,
                         CPU_INT08U          pct_mgmt,
                         CPU_INT08U          pct_dir,
                         FS_FLAGS            mode,
                         FS_ERR             *p_err)
```

File	Called from	Code enabled by
fs_vol.c	Application	FS_CFG_CACHE_EN

Assign cache to a volume.

Arguments

`name_vol` Volume name.

`p_cache_api` Pointer to: (a) cache API to use; OR (b) NULL, if default cache API should be used.

`p_cache_data` Pointer to cache data.

`size` Size, in bytes, of cache buffer.

`pct_mgmt` Percent of cache buffer dedicated to management sectors.

`pct_dir` Percent of cache buffer dedicated to directory sectors.

`mode` Cache mode

`FS_VOL_CACHE_MODE_WR_THROUGH`
`FS_VOL_CACHE_MODE_WR_BACK`
`FS_VOL_CACHE_MODE_RD`

`p_err` Pointer to variable that will receive return error code from this function:

`FS_ERR_NONE`	Cache created.
`FS_ERR_NAME_NULL`	‘name_vol’ passed a NULL pointer.
`FS_ERR_VOL_NOT_OPEN`	Volume not open.
`FS_ERR_NULL_PTR`	‘p_cache_data’ passed a NULL pointer.
`FS_ERR_CACHE_INVALID_MODE`	Mode specified invalid
`FS_ERR_CACHE_INVALID_SEC_TYPE`	Sector type sepecified invalid.
`FS_ERR_CACHE_TOO_SMALL`	Size specified too small for cache.

Returned Value

None.

Notes/Warnings

None.

FSVol_CacheInvalidate ()

```
void  FSVol_CacheInvalidate (CPU_CHAR  *name_vol,
                             FS_ERR    *p_err)
```

File	Called from	Code enabled by
fs_vol.c	Application	FS_CFG_CACHE_EN

Invalidate cache on a volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will receive return error code from this function:

`FS_ERR_NONE`	Cache created.
`FS_ERR_NAME_NULL`	'name_vol' passed a NULL pointer.
`FS_ERR_DEV_CHNGD`	Device has changed.
`FS_ERR_VOL_NO_CACHE`	No cache assigned to volume.
`FS_ERR_VOL_NOT_OPEN`	Volume not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume not mounted.

Returned Value

None.

Notes/Warnings

None.

FSVol_CacheFlush ()

```
void  FSVol_CacheFlush (CPU_CHAR  *name_vol,
                        FS_ERR    *p_err)
```

File	Called from	Code enabled by
fs_vol.c	Application	FS_CFG_CACHE_EN

Flush cache on a volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will receive return error code from this function:

`FS_ERR_NONE`	Cache created.
`FS_ERR_NAME_NULL`	'name_vol' passed a NULL pointer.
`FS_ERR_DEV_CHNGD`	Device has changed.
`FS_ERR_VOL_NO_CACHE`	No cache assigned to volume.
`FS_ERR_VOL_NOT_OPEN`	Volume not open.
`FS_ERR_VOL_NOT_MOUNTED`	Volume not mounted.
`FS_ERR_DEV_INVALID_SEC_NBR`	Sector start or count invalid.
`FS_ERR_DEV_INVALID_lOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout error.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.

Returned Value

None.

Notes/Warnings

None.

A.10 NOR DRIVER FUNCTIONS

```
void
FSDev_NOR_LowFmt          (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_LowMount        (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_LowUnmount      (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_LowCompact      (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_LowDefrag       (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_PhyRd           (CPU_CHAR     *name_dev,
                           void         *p_dest,
                           CPU_INT32U    start,
                           CPU_INT32U    cnt,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_PhyWr           (CPU_CHAR     *name_dev,
                           void         *p_src,
                           CPU_INT32U    start,
                           CPU_INT32U    cnt,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_PhyEraseBlk     (CPU_CHAR     *name_dev,
                           CPU_INT32U    start,
                           CPU_INT32U    size,
                           FS_ERR       *p_err);
```

```
void
FSDev_NOR_PhyEraseChip    (CPU_CHAR     *name_dev,
                           FS_ERR       *p_err);
```

FSDev_NOR_LowFmt()

```
void  FSDev_NOR_LowFmt (CPU_CHAR  *name_dev,
                        FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Low-level format a NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device low-level formatted successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Low-level formating associates physical areas (sectors) of the device with logical sector numbers. A NOR medium MUST be low-level formatted with this driver prior to access by the high-level file system, a requirement which the device module enforces.

FSDev_NOR_LowMount()

```
void  FSDev_NOR_LowMount (CPU_CHAR  *name_dev,
                          FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Low-level mount a NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device low-level mounted successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Low-level mounting parses the on-device structure, detecting the presence of a valid low-level format. If `FS_ERR_DEV_INVALID_LOW_FMT` is returned, the device is NOT low-level formatted.

FSDev_NOR_LowUnmount()

```
void  FSDev_NOR_LowUnmount (CPU_CHAR  *name_dev,
                            FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Low-level unmount a NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device low-level unmounted successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Low-level unmounting clears software knowledge of the on-disk structures, forcing the device to again be low-level mounted or formatted prior to further use.

FSDev_NOR_LowCompact()

```
void  FSDev_NOR_LowCompact (CPU_CHAR  *name_dev,
                            FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Low-level compact a NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device low-level compacted successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Compacting groups sectors containing high-level data into as few blocks as possible. If an image of a file system is to be formed for deployment, to be burned into chips for production, then it should be compacted after all files and directories are created.

FSDev_NOR_LowDefrag()

```
void  FSDev_NOR_LowDefrag (CPU_CHAR  *name_dev,
                           FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Low-level defragment a NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device low-level defragmented successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Defragmentation groups sectors containing high-level data into as few blocks as possible, in order of logical sector. A defragmented file system should have near-optimal access speeds in a read-only environment.

FSDev_NOR_PhyRd()

```
void  FSDev_NOR_PhyRd (CPU_CHAR    *name_dev,
                       void        *p_dest,
                       CPU_INT32U   start,
                       CPU_INT32U   cnt,
                       FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Read from a NOR device and store data in buffer.

Arguments

`name_dev` Device name (see Note).

`p_dest` Pointer to destination buffer.

`start` Start address of read (relative to start of device).

`cnt` Number of octets to read.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Octets read successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_dest` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device.
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

FSDev_NOR_PhyWr()

```
void  FSDev_NOR_PhyWr (CPU_CHAR    *name_dev,
                       void        *p_src,
                       CPU_INT32U   start,
                       CPU_INT32U   cnt,
                       FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Write to a NOR device from a buffer.

Arguments

`name_dev` Device name (see Note).

`p_dest` Pointer to source buffer.

`start` Start address of write (relative to start of device).

`cnt` Number of octets to write.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Octets written successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_src` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device.
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Care should be taken if this function is used while a file system exists on the device, or if the device is low-level formatted. The octet location(s) modified are NOT validated as being outside any existing file system or low-level format information.

During a program operation, only 1 bits can be changed; a 0 bit cannot be changed to a 1. The application MUST know that the octets being programmed have not already been programmed.

FSDev_NOR_PhyEraseBlk()

```
void  FSDev_NOR_PhyEraseBlk (CPU_CHAR    *name_dev,
                             CPU_INT32U   start,
                             CPU_INT32U   size,
                             FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Erase block of NOR device.

Arguments

`name_dev` Device name (see Note).

`start` Start address of block (relative to start of device).

`size` Size of block, in octets.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Block erased successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

Care should be taken if this function is used while a file system exists on the device, or if the device is low-level formatted. The erased block is NOT validated as being outside any existing file system or low-level format information.

FSDev_NOR_PhyEraseChip()

```
void  FSDev_NOR_PhyEraseChip (CPU_CHAR  *name_dev,
                              FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nor.c	Application	N/A

Erase entire NOR device.

Arguments

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	Device erased successfully.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_INVALID_LOW_FMT`	Device needs to be low-level formatted.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a NOR device (e.g., “nor:0:”).

This function should NOT be used while a file system exists on the device, or if the device is low-level formatted, unless the intent is to destroy all existing information.

A.11 SD/MMC DRIVER FUNCTIONS

```
void
FSDev_SD_Card_QuerySD (CPU_CHAR         *name_dev,
                       FS_DEV_SD_INFO   *p_info,
                       FS_ERR           *p_err);
```

```
void
FSDev_SD_SPI_QuerySD  (CPU_CHAR         *name_dev,
                       FS_DEV_SD_INFO   *p_info,
                       FS_ERR           *p_err);
```

```
void
FSDev_SD_Card_RdCID   (CPU_CHAR         *name_dev,
                       CPU_INT08U       *p_info,
                       FS_ERR           *p_err);
```

```
void
FSDev_SD_SPI_RdCID    (CPU_CHAR         *name_dev,
                       CPU_INT08U       *p_info,
                       FS_ERR           *p_err);
```

```
void
FSDev_SD_Card_RdCSD   (CPU_CHAR         *name_dev,
                       CPU_INT08U       *p_info,
                       FS_ERR           *p_err);
```

```
void
FSDev_SD_SPI_RdCSD   (CPU_CHAR          *name_dev,
                      CPU_INT08U        *p_info,
                      FS_ERR            *p_err);
```

FSDev_SD_xxx_QuerySD()

```
void  FSDev_SD_Card_QuerySD (CPU_CHAR        *name_dev,
                             FS_DEV_SD_INFO  *p_info,
                             FS_ERR          *p_err);

void  FSDev_SD_SPI_QuerySD (CPU_CHAR        *name_dev,
                            FS_DEV_SD_INFO  *p_info,
                            FS_ERR          *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card.c, *fs_dev_sd_spi.c*	Application	N/A

Get low-level information abou SD/MMC card.

Arguments

`name_dev` Device name (see Note).

`p_info` Pointer to structure that will receive SD/MMC card information.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	SD/MMC info obtained.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_info` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

The device MUST be a SD/MMC device; (for `FSDev_SD_Card_QuerySD()`, e.g., “sdcard:0:”; for `FSDev_SD_SPI_QuerySD()`, e.g., “sd:0:”).

FSDev_SD_xxx_RdCID()

```
void  FSDev_SD_Card_RdCID (CPU_CHAR    *name_dev,
                           CPU_INT08U  *p_info,
                           FS_ERR      *p_err);

void  FSDev_SD_SPI_RdCID (CPU_CHAR    *name_dev,
                          CPU_INT08U  *p_info,
                          FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card.c, *fs_dev_sd_spi.c*	Application	N/A

Read SD/MMC Card ID (CID) register.

Arguments

`name_dev` Device name (see Note #1).

`p_dest` Pointer to 16-byte buffer that will receive SD/MMC Card ID register.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	SD/MMC Card ID register read.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_dest` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

1. The device MUST be a SD/MMC device; (for `FSDev_SD_Card_QuerySD()`, e.g., “sdcard:0:”; for `FSDev_SD_SPI_QuerySD()`, e.g., “sd:0:”).

2. For SD cards, the structure of the CID is defined in the SD Card Association’s “Physical Layer Simplified Specification Version 2.00”, Section 5.1. For MMC cards, the structure of the CID is defined in the JEDEC’s “MultiMediaCard (MMC) Electrical Standard, High Capacity”, Section 8.2.

FSDev_SD_xxx_RdCSD()

```
void  FSDev_SD_Card_RdCSD (CPU_CHAR    *name_dev,
                           CPU_INT08U  *p_info,
                           FS_ERR      *p_err);

void  FSDev_SD_SPI_RdCSD (CPU_CHAR    *name_dev,
                          CPU_INT08U  *p_info,
                          FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card.c, *fs_dev_sd_spi.c*	Application	N/A

Read SD/MMC Card-Specific Data (CSD) register.

Arguments

`name_dev` Device name (see Note #1).

`p_dest` Pointer to 16-byte buffer that will receive SD/MMC Card-Specific Data register.

`p_err` Pointer to variable that will the receive return error code from this function:

`FS_ERR_NONE`	SD/MMC Card-Specific Data register read.
`FS_ERR_NAME_NULL`	Argument `name_dev` passed a NULL pointer.
`FS_ERR_NULL_PTR`	Argument `p_dest` passed a NULL pointer.
`FS_ERR_DEV_INVALID`	Argument `name_dev` specifies an invalid device
`FS_ERR_DEV_NOT_OPEN`	Device is not open.
`FS_ERR_DEV_NOT_PRESENT`	Device is not present.
`FS_ERR_DEV_IO`	Device I/O error.
`FS_ERR_DEV_TIMEOUT`	Device timeout.

Returned Value

None.

Notes/Warnings

1. The device MUST be a SD/MMC device; (for `FSDev_SD_Card_QuerySD()`, e.g., “sdcard:0:”; for `FSDev_SD_SPI_QuerySD()`, e.g., “sd:0:”).

2. For SD cards, the structure of the CSD is defined in the SD Card Association’s “Physical Layer Simplified Specification Version 2.00”, Section 5.3.2 (v1.x and v2.0 standard capacity) or Section 5.3.3. (v2.0 high capacity). For MMC cards, the structure of the CSD is defined in the JEDEC’s “MultiMediaCard (MMC) Electrical Standard, High Capacity”, Section 8.3.

A.12 FAT SYSTEM DRIVER FUNCTIONS

```
void
FS_FAT_JournalOpen  (CPU_CHAR  *name_vol,
                     FS_ERR    *p_err);
```

```
void
FS_FAT_JournalClose (CPU_CHAR  *name_vol,
                     FS_ERR    *p_err);
```

```
void
FS_FAT_JournalStart (CPU_CHAR  *name_vol,
                     FS_ERR    *p_err);
```

```
void
FS_FAT_JournalStop  (CPU_CHAR  *name_vol,
                     FS_ERR    *p_err);
```

```
void
FS_FAT_VolChk       (CPU_CHAR  *name_vol,
                     FS_ERR    *p_err);
```

FS_FAT_JournalOpen()

File	Called from	Code enabled by
fs_fat_journal.c	Application	FS_CFG_FAT_JOURNAL_EN

Open journal on volume.

Arguments

name_vol	Volume name.	
p_err	Pointer to variable that will the receive return error code from this function:	
	FS_ERR_NONE	Journal opened.
	FS_ERR_DEV	Device access error.

Returned Value

None.

Notes/Warnings

None.

FS_FAT_JournalClose()

```
void  FS_FAT_JournalClose (CPU_CHAR  *name_vol,
                           FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_fat_journal.c	Application	FS_CFG_FAT_JOURNAL_EN

Close journal on volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Journal closed.
FS_ERR_DEV	Device access error.

Returned Value

None.

Notes/Warnings

None.

FS_FAT_JournalStart()

```
void  FS_FAT_JournalStart (CPU_CHAR  *name_vol,
                           FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_fat_journal.c	Application	FS_CFG_FAT_JOURNAL_EN

Start journaling on volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Journaling started.
FS_ERR_DEV	Device access error.

Returned Value

None.

Notes/Warnings

None.

FS_FAT_JournalStop()

```
void  FS_FAT_JournalStop (CPU_CHAR  *name_vol,
                          FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_fat_journal.c	Application	FS_CFG_FAT_JOURNAL_EN

Stop journaling on volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Journaling stopped.
FS_ERR_DEV	Device access error.

Returned Value

None.

Notes/Warnings

None.

FS_FAT_VolChk()

File	Called from	Code enabled by
fs_fat.c	Application	FS_CFG_FAT_VOL_CHK_EN

Check the file system on a volume.

Arguments

`name_vol` Volume name.

`p_err` Pointer to variable that will the receive return error code from this function:

FS_ERR_NONE	Volume checked without errors.
FS_ERR_NAME_NULL	Argument “name_vol” passed a null pointer.
FS_ERR_DEV	Device access error.
FS_ERR_VOL_NOT_OPEN	Volume not open.
FS_ERR_BUF_NONE_AVAIL	No buffers available.

Returned Value

None.

Notes/Warnings

None.

A.13 BSP FUNCTIONS

```
void
FS_BSP_TimeGet (FS_DATE_TIME  *p_time);
```

```
void
FS_BSP_TimeSet (FS_DATE_TIME  *p_time);
```

FS_BSP_TimeGet()

```
void  FS_BSP_TimeGet (FS_DATE_TIME  *p_time)
```

File	Called from	Code enabled by
fs_bsp.c	FSTime_TimeGet()	N/A

Get current date/time.

Arguments

p_time Pointer to variable that will receive the current date/time.

Returned Value

None.

Notes/Warnings

If no date/time can be gotten, the members of p_time SHOULD be zeroed.

Only p_time members Year, Month, Day, Hour, Minute and Second MUST be set. DayOfWeek and DayOfYear will be calculated by FSTime_TimeGet(); any values assigned to those members by this function will be discarded.

Any date/time software or hardware (e.g., RTC) MUST be initialized by the application before it calls FS_Init().

FS_BSP_TimeSet()

```
void  FS_BSP_TimeSet (FS_DATE_TIME  *p_time)
```

File	Called from	Code enabled by
fs_bsp.c	`FSTime_TimeSet()`	N/A

Set current date/time.

Arguments

`p_time` Pointer to date/time to set.

Returned Value

None.

Notes/Warnings

This function is optional; it need only be implemented if the application calls `FSTime_TimeSet()`.

All members of `p_time` are valid, including `DayOfWeek` and `DayOfYear`.

Any date/time software or hardware (e.g., RTC) MUST be initialized by the application before it calls `FS_Init()`.

Appendix

B

µC/FS Error Codes

This appendix provides a brief explanation of µC/FS error codes defined in *fs_err.h*. Any error codes not listed here may be searched in *fs_err.h* for both their numerical value and usage.

B.01 SYSTEM ERROR CODES

FS_ERR_NONE	No error.
FS_ERR_INVALID_ARG	Invalid argument.
FS_ERR_INVALID_CFG	Invalid configuration.
FS_ERR_INVALID_CHKSUM	Invalid checksum.
FS_ERR_INVALID_LEN	Invalid length.
FS_ERR_INVALID_TIME	Invalid date/time.
FS_ERR_INVALID_TIMESTAMP	Invalid timestamp.
FS_ERR_INVALID_TYPE	Invalid object type.
FS_ERR_MEM_ALLOC	Mem could not be alloc'd.
FS_ERR_NULL_ARG	Arg(s) passed NULL val(s).
FS_ERR_NULL_PTR	Ptr arg(s) passed NULL ptr(s).
FS_ERR_OS	OS err.
FS_ERR_OVF	Value too large to be stored in type.
FS_ERR_EOF	EOF reached.
FS_ERR_WORKING_DIR_NONE_AVAIL	No working dir avail.
FS_ERR_WORKING_DIR_INVALID	Working dir invalid.

B.02 BUFFER ERROR CODES

FS_ERR_BUF_NONE_AVAIL	No buffer available.

B.03 CACHE ERROR CODES

FS_ERR_CACHE_INVALID_MODE	Mode specified invalid.
FS_ERR_CACHE_INVALID_SEC_TYPE	Device already open.
FS_ERR_CACHE_TOO_SMALL	Device has changed.

B.04 DEVICE ERROR CODES

FS_ERR_DEV	Device access error.
FS_ERR_DEV_ALREADY_OPEN	Device already open.
FS_ERR_DEV_CHNGD	Device has changed.
FS_ERR_DEV_FIXED	Device is fixed (cannot be closed).
FS_ERR_DEV_FULL	Device is full (no space could be allocated).
FS_ERR_DEV_INVALID	Invalid device.
FS_ERR_DEV_INVALID_CFG	Invalid dev cfg.
FS_ERR_DEV_INVALID_ECC	Invalid ECC.
FS_ERR_DEV_INVALID_IO_CTRL	I/O control invalid.
FS_ERR_DEV_INVALID_LOW_FMT	Low format invalid.
FS_ERR_DEV_INVALID_LOW_PARAMS	Invalid low-level device parameters.
FS_ERR_DEV_INVALID_MARK	Invalid mark.
FS_ERR_DEV_INVALID_NAME	Invalid device name.
FS_ERR_DEV_INVALID_OP	Invalid operation.
FS_ERR_DEV_INVALID_SEC_NBR	Invalid device sec nbr.
FS_ERR_DEV_INVALID_SEC_SIZE	Invalid device sec size.
FS_ERR_DEV_INVALID_SIZE	Invalid device size.
FS_ERR_DEV_INVALID_UNIT_NBR	Invalid device unit nbr.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_NONE_AVAIL	No device avail.
FS_ERR_DEV_NOT_OPEN	Device not open.
FS_ERR_DEV_NOT_PRESENT	Device not present.
FS_ERR_DEV_TIMEOUT	Device timeout.
FS_ERR_DEV_UNIT_NONE_AVAIL	No unit avail.
FS_ERR_DEV_UNIT_ALREADY_EXIST	Unit already exists.
FS_ERR_DEV_UNKNOWN	Unknown.
FS_ERR_DEV_VOL_OPEN	Vol open on dev.

B.05 DEVICE DRIVER ERROR CODES

FS_ERR_DEV_DRV_ALREADY_ADDED	Device driver already added.
FS_ERR_DEV_DRV_INVALID_NAME	Invalid device driver name.
FS_ERR_DEV_DRV_NO_TBL_POS_AVAIL	No pos available in device driver table.

B.06 DIRECTORY ERROR CODES

FS_ERR_DIR_ALREADY_OPEN	Directory already open.
FS_ERR_DIR_DIS	Directory module disabled.
FS_ERR_DIR_FULL	Directory is full.
FS_ERR_DIR_NONE_AVAIL	No directory avail.
FS_ERR_DIR_NOT_OPEN	Directory not open.

B.07 ECC ERROR CODES

FS_ERR_ECC_CORRECTABLE	Correctable ECC error.
FS_ERR_ECC_UNCORRECTABLE	Uncorrectable ECC error.

B.08 ENTRY ERROR CODES

FS_ERR_ENTRIES_SAME	Paths specify same file system entry.
FS_ERR_ENTRIES_TYPE_DIFF	Paths do not both specify files OR directories.
FS_ERR_ENTRIES_VOLS_DIFF	Paths specify file system entries on different vols.
FS_ERR_ENTRY_CORRUPT	File system entry is corrupt.
FS_ERR_ENTRY_EXISTS	File system entry exists.
FS_ERR_ENTRY_INVALID	File system entry invalid.
FS_ERR_ENTRY_NOT_DIR	File system entry NOT a directory.
FS_ERR_ENTRY_NOT_EMPTY	File system entry NOT empty.
FS_ERR_ENTRY_NOT_FILE	File system entry NOT a file.
FS_ERR_ENTRY_NOT_FOUND	File system entry NOT found.
FS_ERR_ENTRY_PARENT_NOT_FOUND	Entry parent NOT found.
FS_ERR_ENTRY_PARENT_NOT_DIR	Entry parent NOT a directory.
FS_ERR_ENTRY_RD_ONLY	File system entry marked read-only.
FS_ERR_ENTRY_ROOT_DIR	File system entry is a root directory.

B.09 FILE ERROR CODES

FS_ERR_FILE_ALREADY_OPEN	File already open.
FS_ERR_FILE_BUF_ALREADY_ASSIGNED	Buf already assigned.
FS_ERR_FILE_ERR	Error indicator set on file.
FS_ERR_FILE_INVALID_ACCESS_MODE	Access mode is specified invalid.
FS_ERR_FILE_INVALID_ATTRIB	Attributes are specified invalid.
FS_ERR_FILE_INVALID_BUF_MODE	Buf mode is specified invalid or unknown.
FS_ERR_FILE_INVALID_BUF_SIZE	Buf size is specified invalid.
FS_ERR_FILE_INVALID_DATE_TIME	Date/time is specified invalid.
FS_ERR_FILE_INVALID_DATE_TIME_FLAG	Date/time flag is specified invalid.
FS_ERR_FILE_INVALID_NAME	Name is specified invalid.
FS_ERR_FILE_INVALID_ORIGIN	Origin is specified invalid or unknown.
FS_ERR_FILE_INVALID_OFFSET	Offset is specified invalid.
FS_ERR_FILE_INVALID_FILES	Invalid file arguments.
FS_ERR_FILE_INVALID_OP	File operation invalid.
FS_ERR_FILE_INVALID_OP_SEQ	File operation sequence invalid.
FS_ERR_FILE_INVALID_POS	File position invalid.
FS_ERR_FILE_LOCKED	File locked.
FS_ERR_FILE_NONE_AVAIL	No file available.
FS_ERR_FILE_NOT_OPEN	File NOT open.
FS_ERR_FILE_NOT_LOCKED	File NOT locked.
FS_ERR_FILE_OVF	File size overflowed max file size.
FS_ERR_FILE_OVF_OFFSET	File offset overflowed max file offset.

B.10 NAME ERROR CODES

FS_ERR_NAME_BASE_TOO_LONG	Base name too long.
FS_ERR_NAME_EMPTY	Name empty.
FS_ERR_NAME_EXT_TOO_LONG	Extension too long.
FS_ERR_NAME_INVALID	Invalid file name or path.
FS_ERR_NAME_MIXED_CASE	Name is mixed case.
FS_ERR_NAME_NULL	Name ptr arg(s) passed NULL ptr(s).

FS_ERR_NAME_PATH_TOO_LONG	Entry path is too long.
FS_ERR_NAME_BUF_TOO_SHORT	Buffer for name is too short.

B.11 PARTITION ERROR CODES

FS_ERR_PARTITION_INVALID	Partition invalid.
FS_ERR_PARTITION_INVALID_NBR	Partition nbr specified invalid.
FS_ERR_PARTITION_INVALID_SIG	Partition sig invalid.
FS_ERR_PARTITION_INVALID_SIZE	Partition size invalid.
FS_ERR_PARTITION_MAX	Max nbr partitions have been created in MBR.
FS_ERR_PARTITION_NOT_FINAL	Prev partition is not final partition.
FS_ERR_PARTITION_NOT_FOUND	Partition NOT found.
FS_ERR_PARTITION_ZERO	Partition zero.

B.12 POOLS ERROR CODES

FS_ERR_POOL_EMPTY	Pool is empty.
FS_ERR_POOL_FULL	Pool is full.
FS_ERR_POOL_INVALID_BLK_ADDR	Block not found in used pool pointers.
FS_ERR_POOL_INVALID_BLK_IN_POOL	Block found in free pool pointers.
FS_ERR_POOL_INVALID_BLK_IX	Block index invalid.
FS_ERR_POOL_INVALID_BLK_NBR	Number blocks specified invalid.
FS_ERR_POOL_INVALID_BLK_SIZE	Block size specified invalid.

B.13 FILE SYSTEM ERROR CODES

FS_ERR_SYS_TYPE_NOT_SUPPORTED	File sys type not supported.
FS_ERR_SYS_INVALID_SIG	Sec has invalid OR illegal sig.
FS_ERR_SYS_DIR_ENTRY_PLACE	Dir entry could not be placed.
FS_ERR_SYS_DIR_ENTRY_NOT_FOUND	Dir entry not found.
FS_ERR_SYS_DIR_ENTRY_NOT_FOUND_YET	Dir entry not found (yet).
FS_ERR_SYS_SEC_NOT_FOUND	Sec not found.
FS_ERR_SYS_CLUS_CHAIN_END	Cluster chain ended.
FS_ERR_SYS_CLUS_CHAIN_END_EARLY	Cluster chain ended before number clusters traversed.
FS_ERR_SYS_CLUS_INVALID	Cluster invalid.
FS_ERR_SYS_CLUS_NOT_AVAIL	Cluster not avail.
FS_ERR_SYS_SFN_NOT_AVAIL	SFN is not avail.
FS_ERR_SYS_LFN_ORPHANED	LFN entry orphaned.

B.14 VOLUME ERROR CODES

FS_ERR_VOL_INVALID_NAME	Invalid volume name.
FS_ERR_VOL_INVALID_SIZE	Invalid volume size.
FS_ERR_VOL_INVALID_SEC_SIZE	Invalid volume sector size.
FS_ERR_VOL_INVALID_CLUS_SIZE	Invalid volume cluster size.
FS_ERR_VOL_INVALID_OP	Volume operation invalid.
FS_ERR_VOL_INVALID_SEC_NBR	Invalid volume sector number.
FS_ERR_VOL_INVALID_SYS	Invalid file system on volume.
FS_ERR_VOL_NO_CACHE	No cache assigned to volume.

FS_ERR_VOL_NONE_AVAIL	No vol avail.
FS_ERR_VOL_NONE_EXIST	No vols exist.
FS_ERR_VOL_NOT_OPEN	Vol NOT open.
FS_ERR_VOL_NOT_MOUNTED	Vol NOT mounted.
FS_ERR_VOL_ALREADY_OPEN	Vol already open.
FS_ERR_VOL_FILES_OPEN	Files open on vol.
FS_ERR_VOL_DIRS_OPEN	Dirs open on vol.
FS_ERR_JOURNAL_ALREADY_OPEN	Journal already open.
FS_ERR_JOURNAL_CFG_CHANGED	File system suite cfg changed since log created.
FS_ERR_JOURNAL_FILE_INVALID	Journal file invalid.
FS_ERR_JOURNAL_FULL	Journal full.
FS_ERR_JOURNAL_LOG_INVALID_ARG	Invalid arg read from journal log.
FS_ERR_JOURNAL_LOG_INCOMPLETE	Log not completely entered in journal.
FS_ERR_JOURNAL_LOG_NOT_PRESENT	Log not present in journal.
FS_ERR_JOURNAL_NOT_OPEN	Journal not open
FS_ERR_JOURNAL_NOT_REPLAYING	Journal not being replayed.
FS_ERR_JOURNAL_NOT_STARTED	Journaling not started.
FS_ERR_JOURNAL_NOT_STOPPED	Journaling not stopped.
FS_ERR_VOL_LABEL_INVALID	Volume label is invalid.
FS_ERR_VOL_LABEL_NOT_FOUND	Volume label was not found.
FS_ERR_VOL_LABEL_TOO_LONG	Volume label is too long.

Appendix

C

µC/FS Types and Structures

Your application may need to access or populate the types and structures described in this appendix. Each of the user-accessible structures is presented in alphabetical order. The following information is provided for each entry:

- A brief description of the type or structure.
- The definition of the type or structure.
- The filename of the source code.
- A description of the meaning of the type or the members of the structure.
- Specific notes and warnings regarding use of the type.

FS_CFG

```
typedef  struct  fs_cfg {
    FS_QTY       DevCnt;
    FS_QTY       VolCnt;
    FS_QTY       FileCnt;
    FS_QTY       DirCnt;
    FS_QTY       BufCnt;
    FS_QTY       DevDrvCnt;
    FS_SEC_SIZE  MaxSecSize;
} FS_CFG;
```

File	Used for
fs.h	First argument of `FS_Init()`

A pointer to a `FS_CFG` structure is the argument of `FS_Init()`. It configures the number of devices, files and other objects in the file system suite.

Members

`DevCnt` — The maximum number of devices that can be open simultaneously. MUST be greater than or equal to 1.

`VolCnt` — The maximum number of volumes that can be open simultaneously. MUST be greater than or equal to 1.

`FileCnt` — The maximum number of files that can be open simultaneously. MUST be greater than or equal to 1.

`DirCnt` — Maximum number of directories that can be open simultaneously. If `DirCnt` is 0, the directory module functions will be blocked after successful initialization, and the file system will operate as if compiled with directory support disabled. If directory support is disabled, `DirCnt` is ignored; otherwise, if directories will be used, `DirCnt` should be greater than or equal to 1.

`BufCnt` — Maximum number of buffers that can be used successfully. The minimum necessary `BufCnt` can be calculated from the number of volumes:

```
BufCnt >= VolCnt * 2
```

If `FSEntry_Copy()` or `FSEntry_Rename()` is used, then up to one additional buffer for each volume may be necessary.

`DevDrvCnt` — Maximum number of device drivers that can be added. It MUST be greater than or equal to 1.

`MaxSecSize` — Maximum sector size, in octets. It must be 512, 1024, 2048 or 4096. No device with a sector size larger than `MaxSecSize` can be opened.

Notes

None.

FS_DATE_TIME

```
typedef  struct  fs_date_time {
    CPU_INT08U  Second;
    CPU_INT08U  Minute;
    CPU_INT08U  Hour;
    CPU_INT08U  Day;
    CPU_INT08U  Month;
    CPU_INT08U  Year;
    CPU_INT08U  DayOfWeek;
    CPU_INT08U  DayOfYear;
} FS_DATE_TIME;
```

File	Used for
fs_time.h	Second argument of `FSEntry_TimeSet()`; First argument of `FSTime_GetTime()`, `FSTime_SetTime()`, `FSTime_Time_to_Str()`, `FSTime_Time_to_TS()`; Second argument of `FSTime_TS_to_Time()`

Holds the date/time of a file or directory.

Members

`Second` The second of the minute as an integer between 0 and 59 (inclusive).

`Minute` The minute of the hour as an integer between 0 and 59 (inclusive).

`Hour` The hour of the day as an integer between 0 and 23 (inclusive).

`Day` The day of the month as an integer between 1 and 28, 29, 30 or 31 (inclusive).

`Month` The month of the year as an integer between 0 and 11 (inclusive), where 0 = January and 11 = December.

`Year` The years since 1900.

`DayOfWeek` The day of the week between 0 and 6 (inclusive), Sunday = 0.

`DayOfYear` The day of the year between 1 and 365 or 366 (inclusive), January 1 = 1.

Notes

All members of a date/time will be zero if and only if it is invalid.

See Chapter 8 for more information about the date/time included with files and directories on FAT file.

FS_DEV_INFO

```
typedef  struct  fs_dev_info {
    FS_STATE     State;
    FS_SEC_QTY   Size;
    FS_SEC_SIZE  SecSize;
} FS_DEV_INFO;
```

File	Used for
fs_dev.h	Second argument of `FSDev_Query()`

Receives information about a device.

Members

`State` The device state:

`FS_DEV_STATE_OPEN`	Device is open, but not present.
`FS_DEV_STATE_PRESENT`	Device is present, but not low-level formatted.
`FS_DEV_STATE_LOW_FMT_VALID`	Device is low-level formatted.

`Size` The number of sectors on the device.

`SecSize` The size of each device sector.

Notes

None.

FS_DEV_NOR_CFG

```
typedef  struct  fs_dev_nor_cfg {
    CPU_ADDR             AddrBase;
    CPU_INT08U           RegionNbr;

    CPU_ADDR             AddrStart;
    CPU_INT32U           DevSize;
    FS_SEC_SIZE          SecSize;
    CPU_INT08U           PctRsvd;
    CPU_INT16U           EraseCntDiffTh;

    FS_DEV_NOR_PHY_API  *PhyPtr;

    CPU_INT08U           BusWidth;
    CPU_INT08U           BusWidthMax;
    CPU_INT08U           PhyDevCnt;
    CPU_INT32U           MaxClkFreq;
} FS_DEV_NOR_CFG;
```

File	Used for
fs_dev_nor.h	Second argument of `FSDev_Open()` (when opening a NOR device)

Configures the properties of a NOR device that will be opened. A pointer to this structure is passed as the second argument of `FSDev_Open()` for a NOR device.

Members

`AddrBase` MUST specify
1. … the base address of the NOR flash memory, for a parallel NOR.
2. … `0x00000000` for a serial NOR.

`RegionNbr` MUST specify the block region which will be used for the file system area. Block regions are enumerated by the physical-layer driver; for more information, see the physical-layer driver header file. (on monolithic devices, devices with only one block region, this MUST be 0).

`AddrStart` MUST specify
1. … the absolute start address of the file system area in the NOR flash memory, for a paralel NOR.
2. … the offset of the start of the file system in the NOR flash, for a serial NOR.

The address specified by `AddrStart` MUST lie within the region `RegionNbr`.

`DevSize` MUST specify the number of octets that will belong to the file system area.

`SecSize` MUST specify the sector size for the NOR flash (either 512, 1024, 2048 or 4096).

`PctRsvd` MUST specify the percentage of sectors on the NOR flash that will be reserved for extra-file system storage (to improve efficiency). This value must be between 5% and 35%, except if 0 is specified whereupon the default will be used (10%).

`EraseCntDiffTh` MUST specify the difference between minimum and maximum erase counts that will trigger passive wear-leveling. This value must be between 5 and 100, except if 0 is

specified whereupon the default will be used (20).

`PhyPtr` MUST point to the appropriate physical-layer driver:

`FSDev_NOR_AMD_1x08`	CFI-compatible parallel NOR implementing AMD command set, 8-bit data bus.
`FSDev_NOR_AMD_1x16`	CFI-compatible parallel NOR implementing AMD command set, 16-bit data bus.
`FSDev_NOR_Intel_1x16`	CFI-compatible parallel NOR implementing Intel command set, 16-bit data bus
`FSDev_NOR_SST39`	SST SST39 Multi-Purpose Flash
`FSDev_NOR_STM25`	ST M25 serial flash
`FSDev_NOR_SST25`	SST SST25 serial flash
Other	User-developed

For a parallel NOR, the bus configuration is specified via `BusWidth`, `BusWidthMax` and `PhyDevCnt`:

`BusWidth`	is the bus width, in bits, between the MCU/MPU and each connected device.
`BusWidthMax`	is the maximum width supported by each connected device.
`PhyDevCnt`	is the number of devices interleaved on the bus.

For a serial flash, the maximum clock frequency is specified via `MaxClkFreq`.

Notes

None.

FS_DEV_RAM_CFG

```
typedef  struct  fs_dev_ram_cfg {
    FS_SEC_SIZE   SecSize;
    FS_SEC_QTY    Size;
    void         *DiskPtr;
} FS_DEV_RAM_CFG;
```

File	Used for
fs_dev_ramdisk.h	Second argument of `FSDev_Open()` (when opening a RAM disk)

Configures the properties of a RAM disk that will be opened. A pointer to this structure is passed as the second argument of `FSDev_Open()` for a RAM disk.

Members

`SecSize` The sector size of RAM disk, either 512, 1024, 2048 or 4096.

`Size` The size of the RAM disk, in sectors.

`DiskPtr` The pointer to the RAM disk.

Notes

None.

FS_DIR_ENTRY (struct dirent)

```
typedef  struct  fs_dirent {
    CPU_CHAR       Name[FS_CFG_MAX_FILE_NAME_LEN + 1];
    FS_ENTRY_INFO  Info;
} FS_DIR_ENTRY;
```

File	Used for
fs_dir.h	Second argument of `fs_readdir_r()` and `FSDir_Rd()`

Receives information about a directory entry.

Members

`Name` The name of the file.

`Info` Entry information. For more information, see Section C.06

Notes

None.

FS_ENTRY_INFO

```
typedef  struct  fs_entry_info {
    FS_FLAGS       Attrib;
    FS_FILE_SIZE   Size;
    FS_TS          DateTimeCreate;
    FS_TS          DateTimeWr;
    FS_SEC_QTY     BlkCnt;
    FS_SEC_SIZE    BlkSize;
} FS_ENTRY_INFO;
```

File	Used for
fs_entry.h	Second argument of `FSEntry_Query()` and `FSFileQuery()`; The `Info` member of `FS_DIR_ENTRY` (`struct fs_dirent`)

Receives information about a file or directory.

Members

`Attrib` The file or directory attributes (see Section 6.02.01).

`Size` The size of the file, in octets.

`DateTimeCreate` The creation timestamp of the file or directory.

`DateTimeWr` The last write (or modification) timestamp of the file or directory.

`BlkCnt` The number of blocks allocated to the file. For a FAT file system, this is the number of clusters occupied by the file data.

`BlkSize` The size of each block allocated in octets. For a FAT file system, this is the size of a cluster.

Notes

None.
.

FS_FAT_SYS_CFG

```
typedef  struct  fs_fat_sys_cfg {
    FS_FAT_CLUS_SIZE  ClusSize;
    FS_FAT_SEC_NBR    RsvdAreaSize;
    CPU_INT16U        RootDirEntryCnt;
    CPU_INT08U        FAT_Type;
    CPU_INT08U        NbrFATs;
} FS_FAT_SYS_CFG;
```

File	Used for
fs_fat_type.h	Second argument of `FSVol_Fmt()` when opening a FAT volume (optional)

A pointer to a `FS_FAT_SYS_CFG` structure may be passed as the second argument of `FSVol_Fmt()`. It configures the properties of the FAT file system that will be created.

Members

`ClusSize` The size of a cluster, in sectors. This should be 1, 2, 4, 8, 16, 32, 64 or 128. The size of a cluster, in bytes, must be less than or equal to 65536, so some of the upper values may be invalid for devices with large sector sizes.

`RsvdAreaSize` The size of the reserved area on the disk, in sectors. For FAT12 and FAT16 volumes, the reserved should be 1 sector; for FAT32 volumes, 32 sectors.

`RootDirEntryCnt` The number of entries in the root directory. This applies only to FAT12 and FAT16 volumes, on which the root directory is a separate area of the file system and is a fixed size. The root directory entry count caps the number of files and directories that can be located in the root directory.

`FAT_Type` The type of FAT. This should be 12 (for FAT12), 16 for (FAT16) or 32 (for FAT32). Ths choice of FAT type must observe restrictions on the maximum number a clusters. A FAT12 file system may have no more than 4085 clusters; a FAT16 file system, no more than 65525.

`NbrFATs` The number of actual FATs (file allocation tables) to create on the disk. The typical value is 2 (one for primary use, a secondary for backup).

Notes

Further restrictions on the members of this structure can be found in Chapter 8.

FS_PARTITION_ENTRY

```
typedef  struct  fs_partition_entry {
    FS_SEC_NBR  Start;
    FS_SEC_QTY  Size;
    CPU_INT08U  Type;
} FS_PARTITION_ENTRY;
```

File	Used for
fs_partition.h	Third argument of `FSDev_PartitionFind()`

Receives information about a partition entry.

Members

`Start` The start sector of partition.

`Size` The size of partition, in sectors.

`Type` The type of data in the partition.

Notes

None.

FS_TS

```
typedef  CPU_INT32U  FS_TS;
```

File	Used for
fs_time.h	First argument of `FSTime_TS_to_Str()` and `FSTime_TS_to_Time()`; Second argument of `FSTime_Time_to_TS()`; The `DateTimeCreate` and `DateTimeWr` members of a `FS_ENTRY_INFO`

Holds the timestamp of a file or directory.

Members

N/A

Notes

The timestamp value is the number of seconds since January 1, 1970. No earlier date has a valid timestamp representation.

The timestamp can encode any date from January 1, 1970 until 2^{32} seconds after that date, slightly more than 136 years later (February 7, 2106).

An invalid timestamp can be detected with:

```
FS_TS  ts;

if (ts == (FS_TS)-1) {
    /* Timestamp invalid */
}
```

FS_VOL_INFO

```
typedef  struct  fs_vol_info {
    FS_STATE     State;
    FS_SEC_QTY   DevSize;
    FS_SEC_SIZE  DevSecSize;
    FS_SEC_QTY   PartitionSize;
    FS_SEC_QTY   VolBadSecCnt;
    FS_SEC_QTY   VolFreeSecCnt;
    FS_SEC_QTY   VolUsedSecCnt;
    FS_SEC_QTY   VolTotSecCnt;
} FS_VOL_INFO;
```

File	Used for
fs_vol.h	Second argument of `FSVol_Query()`

Receives information about a volume.

Members

`State` The volume state:

`FS_VOL_STATE_OPEN`	Volume is open, but not mounted.
`FS_VOL_STATE_MOUNTED`	Volume is mounted.

`DevSize` The number of sectors on the device.

`DevSecSize` The size of each device sector.

`PartitionSize` The number of sectors in the partition.

`VolBadSecCnt` The number of bad sectors on the volume.

`VolFreeSecCnt` The number of free sectors on the volume.

`VolUsedSecCnt` The number of used sectors on the volume.

`VolTotSecCnt` The total number of sectors on the volume.

Notes

None.

Appendix

D

µC/FS Configuration

µC/FS is configurable at compile time via approximately 30 `#defines` in an application's *fs_cfg.h* file. µC/FS uses #defines because they allow code and data sizes to be scaled at compile time based on enabled features. In other words, this allows the ROM and RAM footprints of µC/FS to be adjusted based on your requirements.

Most of the #defines should be configured with the default configuration values. This leaves about a dozen or so values that should be configured with values that may deviate from the default configuration.

D.01 FILE SYSTEM CONFIGURATION

Core file system modules may be selectively disabled.

FS_CFG_SYS_DRV_SEL
FS_CFG_SYS_DRV_SEL selects which file system driver(s) will be included. Currently, there is only one option. When FS_SYS_DRV_SEL_FAT, the FAT system driver will be included.

FS_CFG_CACHE_EN
FS_CFG_CACHE_EN enables (when set to DEF_ENABLED) or disables (when set to DEF_DISABLED) code generation of volume cache functions.

Function	File
FSVol_CacheAssign()	*fs_vol.c*
FSVol_CacheFlush()	*fs_vol.c*
FSVol_CacheInvalidate()	*fs_vol.c*

Table D-1. Cache function exclusion.
These functions are NOT included if FS_CFG_CACHE_EN is DEF_DISABLED.

FS_CFG_API_EN
FS_CFG_API_EN enables (when set to DEF_ENABLED) or disables (when set to DEF_DISABLED) code generation of the POSIX API functions. This API includes functions like fs_fopen() or fs_opendir() which mirror standard POSIX functions like fopen() or opendir().

FS_CFG_DIR_EN
FS_CFG_DIR_EN enables (when set to DEF_ENABLED) or disables (when set to DEF_DISABLED) code generation of directory access functions. When disabled, the functions in the following table will not be available.

Function	File
fs_opendir()	*fs_api.c*
fs_closedir()	*fs_api.c*
fs_readdir_r()	*fs_api.c*
FSDir_Open()	*fs_dir.c*
FSDir_Close()	*fs_dir.c*
FSDir_Rd()	*fs_dir.c*

Table D-2. Directory function exclusion.
These functions are NOT included if FS_CFG_DIR_EN is DEF_DISABLED.

D.02 FEATURE INCLUSION CONFIGURATION

Individual file system features may be selectively disabled.

FS_CFG_FILE_BUF_EN
FS_CFG_BUF_EN enables (when set to DEF_ENABLED) or disables (when set to DEF_DISABLED) code generation of file buffer functions. When disabled, the functions in the following table will not be available.

Function	File
fs_fflush()	*fs_api.c*
fs_setbuf()	*fs_api.c*
fs_setvbuf()	*fs_api.c*
FSFile_BufAssign()	*fs_file.c*
FSFile_BufFlush()	*fs_file.c*

Table D-3. File buffer function exclusion.
These functions are NOT included if FS_CFG_FILE_BUF_EN is DEF_DISABLED.

FS_CFG_FILE_LOCK_EN
FS_CFG_FILE_LOCK_EN enables (when set to DEF_ENABLED) or disables (when set to DEF_DISABLED) code generation of file lock functions. When enabled, a file can be locked across several operations; when disabled, a file is only locked during a single operation and the functions in the following table will not be available.

Function	File
fs_flockfile()	*fs_api.c*
fs_funlockfile()	*fs_api.c*
fs_ftrylockfile()	*fs_api.c*
FSFile_LockGet()	*fs_file.c*
FSFile_LockSet()	*fs_file.c*
FSFile_LockAccept()	*fs_file.c*

Table D-4. File lock function exclusion.
These functions are NOT included if FS_CFG_FILE_LOCK_EN is DEF_DISABLED.

FS_CFG_PARTITION_EN
When FS_CFG_PARTITION_EN is enabled (DEF_ENABLED). volumes can be opened on secondary partitions and partitions can be created. When it is disabled (DEF_DISABLED), volumes can be opened only on the first partition and the functions in the following table will not be available. The function FSDev_PartitionInit(), which initializes the partition structure on a volume, will be included in both configurations.

Function	File
FSDev_GetNbrPartitions()	*fs_dev.c*
FSDev_PartitionAdd()	*fs_dev.c*
FSDev_PartitionFind()	*fs_dev.c*

Table D-5. Partition function exclusion.
These functions are NOT included if FS_CFG_PARTITION_EN is DEF_DISABLED.

FS_CFG_WORKING_DIR_EN

When FS_CFG_WORKING_DIR_EN is enabled (DEF_ENABLED), file system operations can be performed relative to a working directory. When it is disabled (DEF_DISABLED), all file system operations must be performed on absolute paths and the functions in the following table will not be available.

Function	File
fs_chdir()	*fs_api.c*
fs_getcwd()	*fs_api.c*
FS_WorkingDirGet()	*fs.h*
FS_WorkingDirSet()	*fs.h*

Table D-6. Working directory function exclusion.
These functions are NOT included if FS_CFG_WORKING_DIR_EN is DEF_DISABLED.

FS_CFG_UTF8_EN

FS_CFG_UTF8_EN selects whether file names may be specified in UTF-8. When enabled (DEF_ENABLED), file names may be specified in UTF-8; when disabled (DEF_DISABLED), file names must be specified in ASCII.

FS_CFG_RD_ONLY_EN

FS_CFG_RD_ONLY_EN selects whether write access to files, volumes and devices will be possible. When DEF_ENABLED, files, volumes and devices may only be read—code for write operations will not be included and the functions in the following table will not be available.

Function	File
fs_fwrite()	*fs_api.c*
fs_remove()	*fs_api.c*
fs_rename()	*fs_api.c*
fs_mkdir()	*fs_api.c*
fs_truncate()	*fs_api.c*
fs_rmdir()	*fs_api.c*
FSDev_PartitionAdd()	*fs_dev.c*
FSDev_PartitionInit()	*fs_dev.c*
FSDev_Wr()	*fs_dev.c*
FSEntry_AttribSet()	*fs_entry.c*
FSEntry_Copy()	*fs_entry.c*
FSEntry_Create()	*fs_entry.c*
FSEntry_TimeSet()	*fs_entry.c*
FSEntry_Del()	*fs_entry.c*
FSEntry_Rename()	*fs_entry.c*
FSFile_Truncate()	*fs_file.c*
FSFile_Wr()	*fs_file.c*
FSVol_Fmt()	*fs_vol.c*
FSVol_LabelSet()	*fs_vol.c*
FSVol_Wr()	*fs_vol.c*

Table D-7. Read only function exclusion (continued).
These functions are NOT included if FS_CFG_RD_ONLY_EN is DEF_ENABLED.

D.03 NAME RESTRICTION CONFIGURATION

Individual file system features may be selectively disabled.

FS_CFG_MAX_PATH_NAME_LEN
FS_CFG_MAX_PATH_NAME_LEN configures the maximum path name length, in characters (not including the final NULL character). The default value is 260 (the maximum path name length for paths on FAT volumes).

FS_CFG_MAX_FILE_NAME_LEN
FS_CFG_MAX_FILE_NAME_LEN configures the maximum file name length, in characters (not including the final NULL character). The default value is 255 (the maximum file name length for FAT long file names).

FS_CFG_MAX_DEV_DRV_NAME_LEN
FS_CFG_MAX_DEV_DRV_NAME_LEN configures the maximum device driver name length, in characters (not including the final NULL character). The default value is 10.

FS_CFG_MAX_DEV_NAME_LEN
FS_CFG_MAX_DEV_NAME_LEN configures the maximum device name length, in characters (not including the final NULL character). The default value is 15.

FS_CFG_MAX_VOL_NAME_LEN
FS_CFG_MAX_VOL_NAME_LEN configures the maximum volume name length, in characters (not including the final NULL character). The default value is 10.

D.04 DEBUG CONFIGURATION

A fair amount of code in µC/FS has been included to simplify debugging. There are several configuration constants used to aid debugging.

FS_CFG_DBG_MEM_CLR_EN

FS_CFG_DBG_MEM_CLR_EN is used to clear internal file system data structures when allocated or deallocated. When DEF_ENABLED, internal file system data structures will be cleared.

FS_CFG_DBG_WR_VERIFY_EN

FS_CFG_DBG_WR_VERIFY_EN is used verify writes by reading back data. This is a particularly convenient feature while debugging a driver.

D.05 ARGUMENT CHECKING CONFIGURATION

Most functions in µC/FS include code to validate arguments that are passed to it. Specifically, µC/FS checks to see if passed pointers are `NULL`, if arguments are within valid ranges, etc. The following constants configure additional argument checking.

`FS_CFG_ARG_CHK_EXT_EN`
`FS_CFG_ARG_CHK_EXT_EN` allows code to be generated to check arguments for functions that can be called by the user and for functions which are internal but receive arguments from an API that the user can call.

`FS_CFG_ARG_CHK_DBG_EN`
`FS_CFG_ARG_CHK_DBG_EN` allows code to be generated which checks to make sure that pointers passed to functions are not `NULL`, that arguments are within range, etc.:

D.06 FILE SYSTEM COUNTER CONFIGURATION

µC/FS contains code that increments coutners to keep track of statistics such as the number of packets received, the number of packets transmitted, etc. Also, µC/FS contains counters that are incremented when error conditions are detected.

FS_CFG_CTR_STAT_EN
FS_CFG_CTR_STAT_EN determines whether the code and data space used to keep track of statistics will be included. When DEF_ENABLED, statistics counters will be maintained.

FS_CFG_CTR_ERR_EN
FS_CFG_CTR_STAT_EN determines whether the code and data space used to keep track of errors will be included. When DEF_ENABLED, error counters will be maintained.

D.07 FAT CONFIGURATION

Configuration constants can be used to enable/disable features within the FAT file system driver.

FS_FAT_CFG_LFN_EN
FS_FAT_CFG_LFN_EN is used to control whether long file names (LFNs) are supported. When DEF_DISABLED, all file names must be valid 8.3 short file names.

FS_FAT_CFG_FAT12_EN
FS_FAT_CFG_FAT12_EN is used to control whether FAT12 is supported. When DEF_DISABLED, FAT12 volumes can not be opened, nor can a device be formatted as a FAT12 volume.

FS_FAT_CFG_FAT16_EN
FS_FAT_CFG_FAT16_EN is used to control whether FAT16 is supported. When DEF_DISABLED, FAT16 volumes can not be opened, nor can a device be formatted as a FAT16 volume.

FS_FAT_CFG_FAT32_EN
FS_FAT_CFG_FAT32_EN is used to control whether FAT32 is supported. When DEF_DISABLED, FAT32 volumes can not be opened, nor can a device be formatted as a FAT32 volume.

FS_FAT_CFG_JOURNAL_EN
FS_FAT_CFG_JOURNAL_EN selects whether journaling functions will be present. When DEF_ENABLED, journaling functions are present; when DEF_DISABLED, journaling functions are NOT present. If disabled, the functions in Table D-8 will not be available.

Function	File
FS_FAT_JournalOpen()	*fs_fat_journal.c/.h*
FS_FAT_JournalClose()	*fs_fat_journal.c/.h*
FS_FAT_JournalStart()	*fs_fat_journal.c/.h*
FS_FAT_JournalEnd()	*fs_fat_journal.c/.h*

Table D-8. Journaling function exclusion.
These functions are NOT included if FS_FAT_CFG_JOURNAL_EN is DEF_DISABLED.

FS_FAT_CFG_VOL_CHK_EN
FS_FAT_CFG_VOL_CHK_EN selects whether volume check is supported. When DEF_ENABLED, volume check is supported; when DEF_DISABLED, the function FS_FAT_VolChk() will not be available.

FS_FAT_CFG_VOL_CHK_MAX_LEVELS
FS_FAT_CFG_VOL_CHK_MAX_LEVELS specifies the maximum number of directory levels that will be checked by the volume check function. Each level requires an additional 12 bytes stack space.

D.08 SD/MMC SPI CONFIGURATION

FS_DEV_SD_SPI_CFG_CRC_EN

Data blocks received from the card are accompanied by CRCs, as are the blocks transmitted to the card. FS_DEV_SD_SPI_CFG_CRC_EN enables CRC validation by the card, as well as the generation and checking of CRCs. If DEF_ENABLED, CRC generation and checking will be performed.

D.09 TIMESTAMP CONFIGURATION

FS_CFG_GET_TS_FROM_OS

Specifies if the timestamp calculation is done is the file system OS layer (`fs_os.c` file, `FS_OS_GetTS_ms()`) or provided by the application. When set to `DEF_ENABLED`, the timestamp calculation needed in `FS_BSP_GetTS_ms()` will be done in the file system OS layer.

D.10 TRACE CONFIGURATION

The file system debug trace is enabled by `#define`'ing `FS_TRACE_LEVEL` in your application's *app_cfg.h*:

```
#define  FS_TRACE_LEVEL                         TRACE_LEVEL_DBG
```

The valid trace levels are described in the table below. A trace functions should also be defined:

```
#define  FS_TRACE                               printf
```

This should be a `printf`-type function that redirects the trace output to some accessible terminal (for example, the terminal I/O window within your debugger, or a serial port) . When porting a driver to a new platform, this information can be used to debug the fledgling port.

Trace Level	Meaning
TRACE_LEVEL_OFF	No trace.
TRACE_LEVEL_INFO	Basic event information (e.g., volume characteristics).
TRACE_LEVEL_DBG	Debug information.
TRACE_LEVEL_LOG	Event log.

Table D-9. Trace Levels

Appendix

E

Shell Commands

The command line interface is a traditional method for accessing the file system on a remote system, or in a device with a serial port (be that RS-232 or USB). A group of shell commands, derived from standard UNIX equivalents, are available for µC/FS. These may simply expedite evaluation of the file system suite, or become part a primary method of access (or gathering debug information) in your final product.

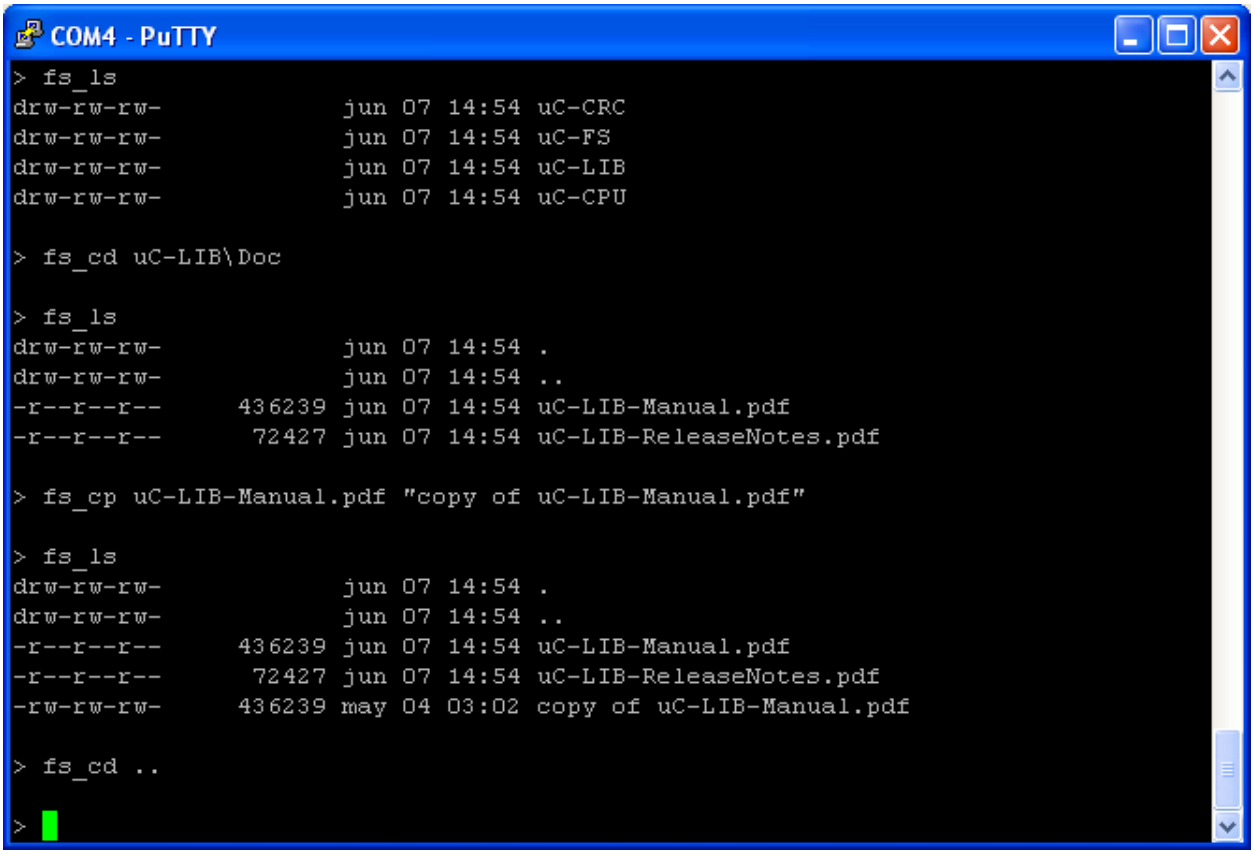


Figure E-1. µC/FS shell command usage

E.01 FILES AND DIRECTORIES

µC/FS with the shell commands (and µC/Shell) is organized into the directory structure shown in Figure E-1. The files constituting the shell commands ares outlined in this section; the generic file-system files, outlined in Section 1.02, are also required.

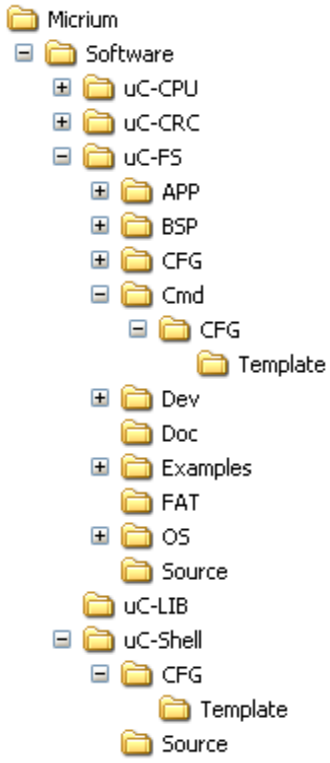


Figure E-1. Directory Structure.

\Micrium\Software\uC-FS \Cmd

fs_shell.* contain the shell commands for µC/FS.

\Micrium\Software\uC-FS\Cmd\Template\Cfg

fs_shell_cfg.h is the template configuration file for the µC/FS shell commands. This file should be copied to your application directory and modified.

\Micrium\Software\uC-Shell

This directory contains µC/Shell, which is used to process the commands. See the µC/Shell user manual for more information.

E.02 USING THE SHELL COMMANDS

To use shell commands, four files, in addition to the generic file system files, must be included in the build:

- *fs_shell.c* (located in the directory specified in Section E.01).
- *fs_ shell.h* (located in the directory specified in Section E.01).
- *shell.c* (located in *\Micrium\Software\uC-Shell\Source*).
- *shell.h* (located in *\Micrium\Software\uC-Shell\Source*).

The file *fs_shell.h* and *shell.h* must also be `#included` in any application or header files initialize µC/Shell or handle shell commands. The shell command configuration file (*fs_shell_cfg.h*) should be copied to your application directory and modified. The following directories must be on the project include path:

- *\Micrium\Software\uC-FS\Cmd*
- *\Micrium\Software\uC-Shell\Source*

µC/Shell with the µC/FS shell commands is initialized in Listing E-1. The file system initialization (`FS_Init()`) function should have previously been called.

Listing E-1. Initializing µC/Shell.

```
CPU_BOOLEAN  App_ShellInit (void)
{
    CPU_BOOLEAN  ok;


    ok = Shell_Init();
    if (ok == DEF_FAIL) {
        return (DEF_FAIL);
    }


    ok = FSShell_Init();
    if (ok == DEF_FAIL) {
        return (DEF_FAIL;
    }

    return (DEF_OK);
}
```

It's assumed that the application will create a task to receive input from a terminal; this task should be written as shown in Listing E-2.

Listing E-2. Executing shell commands & handling shell output.

```
void  App_ShellTask (void *p_arg)
{
    CPU_CHAR          cmd_line[MAX_CMD_LEN];
    SHELL_ERR         err;
    SHELL_CMD_PARAM   cmd_param;
    CPU_CHAR          cwd_path[FS_CFG_FULL_ NAME_LEN + 1u];


                                                /* Init cmd param (see Note #1). */
    Str_Copy(&cwd_path[0], (CPU_CHAR *)"\\");
    cmd_param.pcur_working_dir = (void *)cwd_path[0];
    cmd_param.pout_opt         = (void *)0;

    while (DEF_TRUE) {
        App_ShellIn(cmd_line, MAX_CMD_LEN);       /* Rd  cmd       (see Note #2). */
```

```
                                                    /* Exec cmd       (see Note #3). */
        Shell_Exec(cmd_line, App_ShellOut, &cmd_param, &err);
        switch (err) {
            case SHELL_ERR_CMD_NOT_FOUND:
            case SHELL_ERR_CMD_SEARCH:
            case SHELL_ERR_ARG_TBL_FULL:
                 App_ShellOut("Command not found\r\n", 19, cmd_param.pout_opt);
                 break;

            default:
                 break;
        }
    }
}

/*
*********************************************************************************************
*                                     App_ShellIn()
*********************************************************************************************1*******
*/

CPU_INT16S  App_ShellIn (CPU_CHAR    *pbuf,
                         CPU_INT16U   buf_len)
{
    /* $$$$ Store line from terminal/command line into 'pbuf'; return length of line. */
}

/*
*********************************************************************************************
*                                     App_ShellOut()
*********************************************************************************************1*******
*/

CPU_INT16S  App_ShellOut (CPU_CHAR    *pbuf,
                          CPU_INT16U   buf_len,
                          void        *popt)
{
    /* $$$$ Output 'pbuf' data on terminal/command line;  return nbr bytes tx'd. */
}
```

LE-2(1) The `SHELL_CMD_PARAM` structure that will be passed to `Shell_Exec()` must be initialized. The `pcur_working_dir` member MUST be assigned a pointer to a string of at least `FS_SHELL_CFG_MAX_PATH_LEN` characters. This string must have been initialized to the default working directory path; if the root directory, "\".

LE-2(2) The next command, ending with a newline, should be read from the command line.

LE-2(3) The received command should be executed with `Shell_Exec()`. If the command is a valid command, the appropriate command function will be called. For example, the command "fs_ls" will result in `FSShell_ls()` in *fs_shell.c* being called. `FSShell_ls()` will then print the entries in the working directory to the command line with the output function `App_ShellOut()`, passed as the second argument of `Shell_Exec()`.

E.03 COMMANDS

The supported commands, listed in the table below, are equivalent to the standard UNIX commands of the same names, though the functionality is typically simpler, with few or no special options.

Command	Description
`fs_cat`	Print file contents to the terminal output.
`fs_cd`	Change the working directory.
`fs_cp`	Copy a file.
`fs_date`	Write the date and time to terminal output, or set the system date and time
`fs_df`	Report disk free space.
`fs_ls`	List directory contents.
`fs_mkdir`	Make a directory.
`fs_mkfs`	Format a volume.
`fs_mount`	Mount volume.
`fs_mv`	Move files.
`fs_od`	Dump file contents to terminal output.
`fs_pwd`	Write to terminal output pathname of current working directory.
`fs_rm`	Remove a directory entry.
`fs_rmdir`	Remove a directory.
`fs_touch`	Change file modification time.
`fs_umount`	Unmount volume.
`fs_wc`	Determine the number of newlines, words and bytes in a file.

Table E-1. Commands

Information about each command can be obtained using the help (`-h`) option:

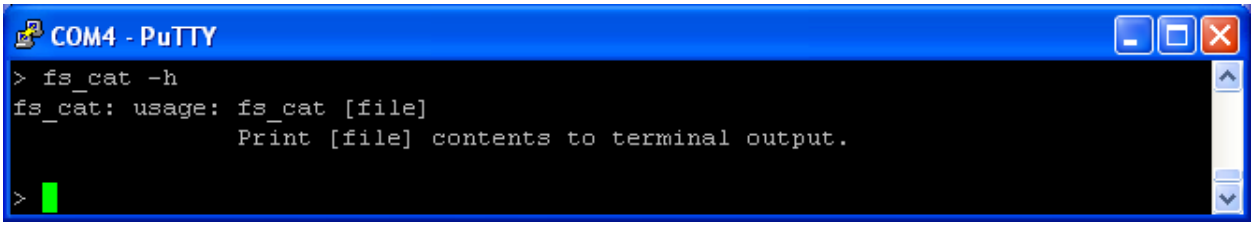


Figure E-2. Help option output

fs_cat

Print file contents to the terminal output.

Usages

```
fs_cat  [file]
```

Arguments

`file`	Path of file to print to terminal output.

Output

File contents, in the ASCII character set. Non-printable/non-space characters are transmitted as full stops ("periods", character code 46). For a more convenient display of binary files use `fs_od`.

Required Configuration

Available only if `FS_SHELL_CFG_CAT_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

fs_cd

Change the working directory.

Usages

```
fs_cd  [dir]
```

Arguments

`dir`	Absolute directory path. OR Path relative to current working directory.

Output

None.

Required Configuration

Available only if `FS_SHELL_CFG_CD_EN` is `DEF_ENABLED`.

Notes/Warnings

The new working directory is formed in three steps:

1. If the argument `dir` begins with the path separator character (slash, `'\'`) or a volume name, it will be interpreted as an absolute directory path and will become the preliminary working directory. Otherwise the preliminary working directory path is formed by the concatenation of the current working directory, a path separator character and `dir`.

2. The preliminary working directory path is then refined, from the first to last path component:

    a. If the component is a ‘dot’ component, it is removed
    b. If the component is a ‘dot dot’ component, and the preliminary working directory path is not NULL, the previous path component is removed. In any case, the ‘dot dot’ component is removed.
    c. Trailing path separator characters are removed, and multiple path separator characters are replaced by a single path separator character.

3. The volume is examined to determine whether the preliminary working directory exists. If it does, it becomes the new working directory. Otherwise, an error is output, and the working directory is unchanged.

fs_cp

Copy a file.

Usages

```
fs_cp  [source_file] [dest_file]
```

```
fs_cp  [source_file] [dest_dir]
```

Arguments

source_file	Source file path.
dest_file	Destination file path.
dest_dir	Destination directory path.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_CP_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

In the first form of this command, neither argument may be an existing directory. The contents of source_file will be copied to a file named dest_file located in the same directory as source_file.

In the second form of this command, the first argument must not be an existing directory and the second argument must be an existing directory. The contents of source_file will be copied to a file with name formed by concatenating dest_dir, a path separator character and the final component of source_file.

fs_date

Write the date and time to terminal output, or set the system date and time.

Usages

```
fs_date
```

```
fs_date  [time]
```

Arguments

`time` If specified, time to set, in the form `mmddhhmmccyy`:

where	the 1st	`mm`	is the month	(1-12)
	the	`dd`	is the day	(1-29, 30 or 31)
	the	`hh`	is the hour	(0-23)
	the 2nd	`mm`	is the minute	(0-59)
	the	`ccyy`	is the year	(1900 or larger)

Output

If no argument, date and time.

Required Configuration

Available only if `FS_SHELL_CFG_DATE_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

Figure E-3. `fs_date` output

fs_df

Report disk free space.

Usages

```
fs_df
```

```
fs_df  [vol]
```

Arguments

`vol` If specified, volume on which to report free space. Otherwise, information about all volumes will be output..

Output

Name, total space, free space and used space of volumes.

Required Configuration

Available only if `FS_SHELL_CFG_DF_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

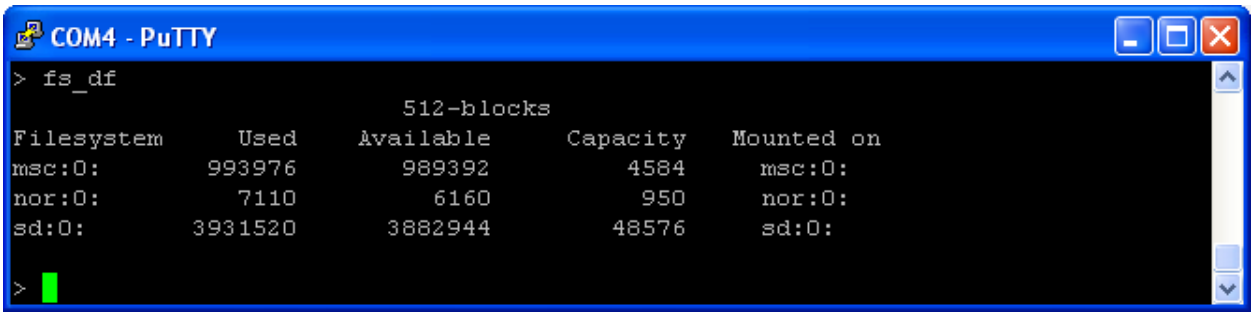


Figure E-4. `fs_df` output

fs_ls

List directory contents.

Usages

```
fs_ls
```

Arguments

None.

Output

List of directory contents.

Required Configuration

Available only if `FS_SHELL_CFG_LS_EN` is `DEF_ENABLED`.

Notes/Warnings

The output resembles the output from the standard UNIX command `ls -l`. See the figure below.

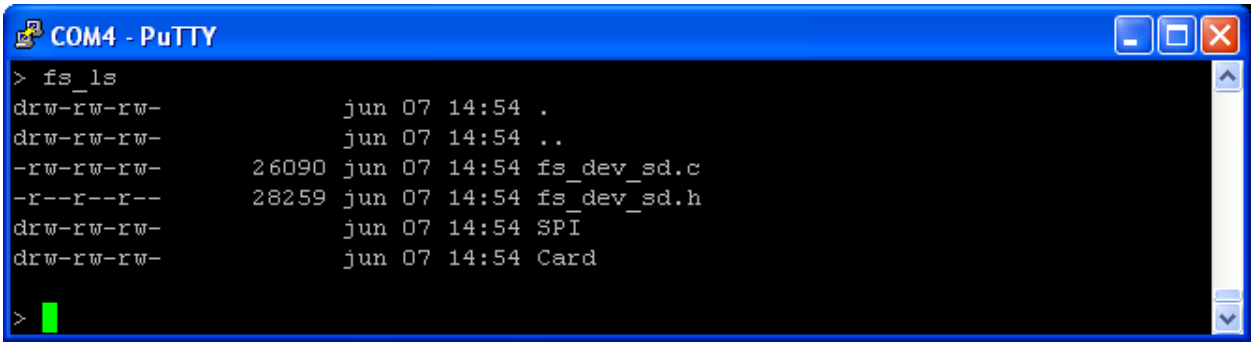


Figure E-5. `fs_ls` output

fs_mkdir

Make a directory.

Usages

```
fs_mkdir  [dir]
```

Arguments

dir	Directory path.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_MKDIR_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

None.

fs_mkfs

Format a volume.

Usages

```
fs_mkfs  [vol]
```

Arguments

vol	Volume name.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_MKFS_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

None.

fs_mount

Mount volume.

Usages

```
fs_mount  [dev]  [vol]
```

Arguments

dev	Device to mount.
vol	Name which will be given to volume.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_MOUNT_EN is DEF_ENABLED.

Notes/Warnings

None.

fs_mv

Move files.

Usages

```
fs_mv  [source_entry] [dest_entry]
```

```
fs_mv  [source_entry] [dest_dir]
```

Arguments

source_entry	Source entry path.
dest_entry	Destination entry path.
dest_dir	Destination directory path.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_MV_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

In the first form of this command, the second argument must not be an existing directory. The file source_entry will be renamed dest_entry.

In the second form of this command, the second argument must be an existing directory. source_entry will be renamed to an entry with name formed by concatenating dest_dir, a path separator character and the final component of source_entry.

In both forms, if source_entry is a directory, the entire directory tree rooted at source_entry will be copied and then deleted. Additionally, both source_entry and dest_entry or dest_dir must specify locations on the same volume.

fs_od

Dump file contents to the terminal output.

Usages

```
fs_od  [file]
```

Arguments

`file`	Path of file to dump to terminal output.

Output

File contents, in hexadecimal form.

Required Configuration

Available only if `FS_SHELL_CFG_OD_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

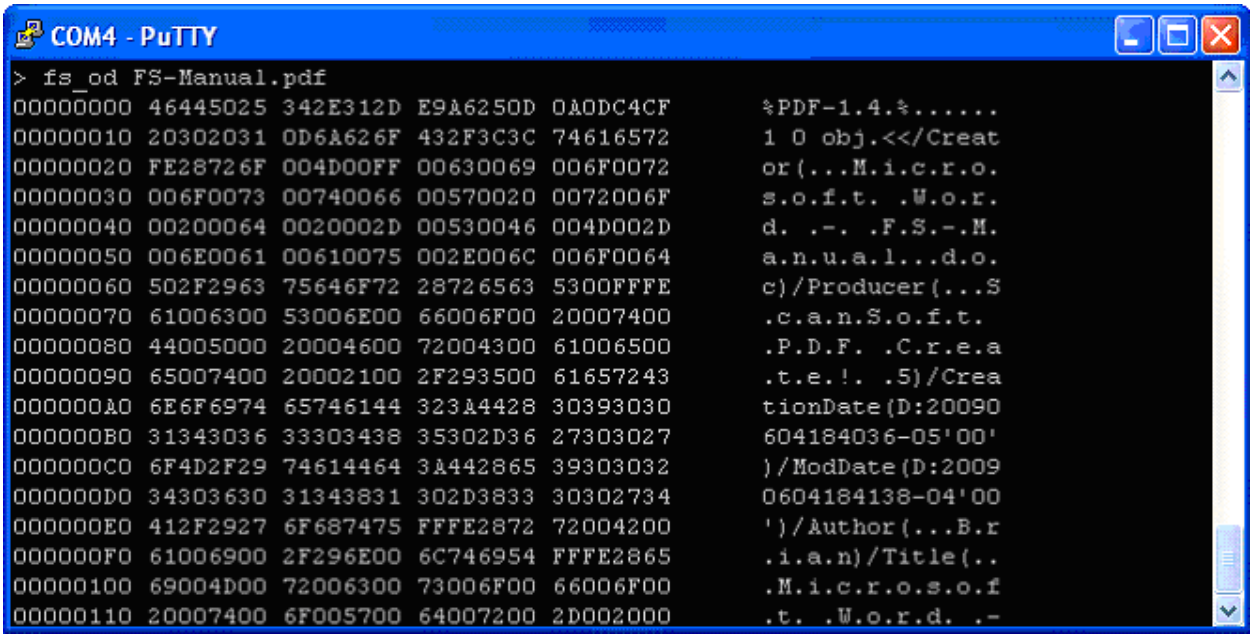


Figure E-6. `fs_od` output

fs_pwd

Write to terminal output pathname of current working directory.

Usages

```
fs_pwd
```

Arguments

None.

Output

Pathname of current working directory..

Required Configuration

Available only if FS_SHELL_CFG_PWD_EN is DEF_ENABLED.

Notes/Warnings

None.

fs_rm

Remove a directory entry.

Usages

```
fs_rm  [entry]
```

Arguments

entry	File or directory path.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_RM_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

None.

fs_rmdir

Remove a directory.

Usages

```
fs_rmdir  [dir]
```

Arguments

dir	Directory path.

Output

None.

Required Configuration

Available only if FS_SHELL_CFG_RMDIR_EN is DEF_ENABLED
and FS_CFG_RD_ONLY_EN is DEF_DISABLED.

Notes/Warnings

None.

fs_touch

Change file modification time.

Usages

```
fs_touch  [file]
```

Arguments

`file`	File path.

Output

None.

Required Configuration

Available only if `FS_SHELL_CFG_TOUCH_EN` is `DEF_ENABLED`
and `FS_CFG_RD_ONLY_EN` is `DEF_DISABLED`.

Notes/Warnings

The file modification time is set to the current time.

fs_umount

Unount volume.

Usages

```
fs_umount  [vol]
```

Arguments

`vol`	Volume to unmount.

Output

None.

Required Configuration

Available only if `FS_SHELL_CFG_UMOUNT_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

fs_wc

Determine the number of newlines, words and bytes in a file.

Usages

```
fs_wc  [file]
```

Arguments

`file`	Path of file to examine.

Output

Number of newlines, words and bytes; equivalent to:

```
printf("%d %d %d %s", newline_cnt, word_cnt, byte_cnt, file);
```

Required Configuration

Available only if `FS_SHELL_CFG_WC_EN` is `DEF_ENABLED`.

Notes/Warnings

None.

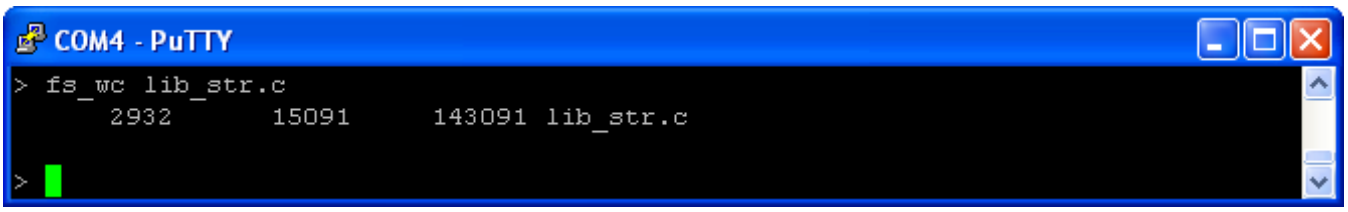


Figure E-7. `fs_wc` output

E.04 CONFIGURATION

Configuration constants can be used to enable/disable features within the µC/FS shell commands.

FS_SHELL_CFG_BUF_LEN
FS_FAT_CFG_BUF_LEN defines the length of the buffer, in octets, used to read/write from files during file access operations. Since this buffer is placed on the task stack, the task stack must be sized appropraitely.

FS_SHELL_CFG_CMD_####_EN
Each FS_FAT_CFG_CMD_####_EN separately enables/disables a particular fs_#### command:

FS_FAT_CFG_CMD_CAT_EN	Enable/disable `fs_cat`.
FS_FAT_CFG_CMD_CD_EN	Enable/disable `fs_cd`.
FS_FAT_CFG_CMD_CP_EN	Enable/disable `fs_cp`.
FS_FAT_CFG_CMD_DF_EN	Enable/disable `fs_df`.
FS_FAT_CFG_CMD_DATE_EN	Enable/disable `fs_date`.
FS_FAT_CFG_CMD_LS_EN	Enable/disable `fs_ls`.
FS_FAT_CFG_CMD_MKDIR_EN	Enable/disable `fs_mkdir`.
FS_FAT_CFG_CMD_MKFS_EN	Enable/disable `fs_mkfs`.
FS_FAT_CFG_CMD_MOUNT_EN	Enable/disable `fs_mount`.
FS_FAT_CFG_CMD_MV_EN	Enable/disable `fs_mv`.
FS_FAT_CFG_CMD_OD_EN	Enable/disable `fs_od`.
FS_FAT_CFG_CMD_PWD_EN	Enable/disable `fs_pwd`.
FS_FAT_CFG_CMD_RM_EN	Enable/disable `fs_rm`.
FS_FAT_CFG_CMD_RMDIR_EN	Enable/disable `fs_rmdir`.
FS_FAT_CFG_CMD_TOUCH_EN	Enable/disable `fs_touch`.
FS_FAT_CFG_CMD_UMOUNT_EN	Enable/disable `fs_umount`.
FS_FAT_CFG_CMD_WC_EN	Enable/disable `fs_wc`.

Appendix

F

References

General

The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2004 Edition
Available at http://www.unix.org/single_unix_specification/
Referred to as "IEEE 1003.1" or "Unix standard" or "POSIX".

ISO/IEC 9899:TC2 Programming Languages – C
Referred to as "ISO/IEC 9899" or "C standard".

FAT

Microsoft Corporation. "FAT32 File System Specification. FAT: General Overview of On-Disk Format." Version 1.03. December 6, 2000.
Microsoft's brief "specification" of the FAT file system, available at

http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

Compact Flash / IDE

CompactFlash Association. "CF+ and CompactFlash Specification Revision 4.1." 02/16/07.
CompactFlash specification.
Available on the the CompactFlash's website http://www.compactflash.org**.**

ANSI. "Information Technology – AT Attachment with Packet Interface – 6 (ATA/ATAPI-6)".
The latest ANSI standard for ATA devices covering a superset of CompactFlash and typical ATA hard disk drive functionality.

SD / MMC

SD Group. "SD Specifications, Part 1: Physical Layer Simplified Specification." Version 2.00. July 27, 2006.

The physical layer specification for SD cards.
Available on the SD card association's website http://www.sdcard.org.

JEDEC Solid State Technology Association. "MultiMediaCard (MMC) Electrical Standard High Capacity)". JESD84-B42. July 2007.
The most recent version of the MMC specification supporting SPI mode, available from the website of JEDEC, which co-developed the technology: http://www.jedec.org.

JEDEC Solid State Technology Association. "Embedded MultiMediaCard (eMMC) eMMC/Card Product Standard, High Capacity, including Reliable Write, Boot, and Sleep Modes". JESD84-A43. November 2007.
The latest version of the MMC specification, specifying eMMC, also available from the website of JEDEC.