µC/FS™

The Embedded File System

User's Manual
V4.04

Micriµm

For the Way Engineers Work

Micriμm
1290 Weston Road, Suite 306
Weston, FL 33326
USA

www.micrium.com

600-uC-FS-001

Micriμm

For the Way Engineers Work

Table of Contents

Chapter 1	Introduction	15
1-1	What Is a File System?	15
1-2	µC/FS	16
1-3	Typical Usages	17
1-4	Why FAT?	18
1-5	Chapter Contents	18
Chapter 2	µC/FS Architecture	22
2-1	Architecture Components	24
2-1-1	Your Application	24
2-1-2	LIB (Libraries)	24
2-1-3	POSIX API Layer	24
2-1-4	FS Layer	24
2-1-5	File System Driver Layer	26
2-1-6	Device Driver Layer	26
2-1-7	CPU Layer	26
2-1-8	RTOS Layer	27
Chapter 3	Directories and Files	28
3-1	Application Code	31
3-2	CPU	33
3-3	Board Support Package (BSP)	34
3-4	µC/CPU, CPU Specific Source Code	35
3-5	µC/LIB, Portable Library Functions	37
3-6	µC/Clk, Time/Calendar Management	38
3-7	µC/CRC, Checksums and Error Correction Codes	40
3-8	µC/FS Platform-Independent Source Code	42
3-9	µC/FS FAT Filesystem Source Code	45
3-10	µC/FS Memory Device Drivers	46

3-11	µC/FS Platform-Specific Source Code	50
3-12	µC/FS OS Abstraction Layer	51
3-13	Summary	52
Chapter 4	Miscellaneous	59
4-1	Nomenclature	59
4-2	µC/FS Device and Volume Names	61
4-3	µC/FS File and Directory Names and Paths	62
4-4	µC/FS Name Lengths	63
4-5	Resource Usage	65
Chapter 5	Devices and Volumes	67
5-1	Device Operations	68
5-2	Using Devices	69
5-3	Using Removable Devices	71
5-4	Partitions	72
5-5	Volume Operations	76
5-6	Using Volumes	77
5-7	Using Volume Cache	79
5-7-1	Choosing Cache Parameters	80
5-7-2	Other Caching & Buffering Mechanisms	82
Chapter 6	POSIX API	83
6-1	Supported Functions	84
6-2	Working Directory Functions	85
6-3	File Access Functions	86
6-3-1	Opening, Reading & Writing Files	87
6-3-2	Getting or Setting the File Position	90
6-3-3	Configuring a File Buffer	91
6-3-4	Diagnosing a File Error	93
6-3-5	Atomic File Operations Using File Lock	93
6-4	Directory Access Functions	94
6-5	Entry Access Functions	96
Chapter 7	Files	97
7-1	File Access Functions	98
7-1-1	Opening Files	99

7-1-2	Getting Information About a File	100
7-1-3	Configuring a File Buffer	101
7-1-4	File Error Functions	102
7-1-5	Atomic File Operations Using File Lock	102
7-2	Entry Access Functions	103
7-2-1	File and Directory Attributes	104
7-2-2	Creating New Files and Directories	105
7-2-3	Deleting Files and Directories	106
Chapter 8	Directories	107
8-1	Directory Access Functions	108
Chapter 9	File Systems: FAT	109
9-1	FAT Architecture	110
9-1-1	FAT12 / FAT16 / FAT32	111
9-1-2	Short and Long File Names	111
9-1-3	Directories and Directory Entries	112
9-1-4	FAT System Driver Architecture	114
9-2	Operations	115
9-2-1	Formatting	115
9-2-2	Disk Check	116
9-2-3	Journaling	117
Chapter 10	Device Drivers	119
10-1	Provided Device Drivers	120
10-1-1	Driver Characterization	121
Chapter 11	IDE/CF Driver	124
11-1	Files and Directories	124
11-2	Using the IDE/CF Driver	125
11-2-1	ATA (True IDE) Communication	128
11-2-2	IDE BSP Overview	131
Chapter 12	Logical Device Driver	133
Chapter 13	MSC Driver	134

13-1	Files and Directories	134
13-2	Using the MSC Driver	135
Chapter 14	NAND Flash Driver	137
14-1	Files and Directories	138
14-2	Driver & Device Characteristics	139
14-3	Using a NAND Device (Software ECC)	140
14-3-1	Driver Architecture	146
14-3-2	Hardware	146
14-3-3	NAND BSP Overview	148
14-4	Physical-Layer Drivers	148
14-4-1	FSDev_NAND_0512x08	149
14-4-2	FSDev_NAND_2048x08, FSDev_NAND_2048x16	149
14-4-3	FSDev_NAND_AT45	150
Chapter 15	NOR Flash Driver	151
15-1	Files and Directories	152
15-2	Driver & Device Characteristics	154
15-3	Using a Parallel NOR Device	156
15-3-1	Driver Architecture	160
15-3-2	Hardware	160
15-3-3	NOR BSP Overview	162
15-4	Using a Serial NOR Device	163
15-4-1	Hardware	164
15-4-2	NOR SPI BSP Overview	165
15-5	Physical-Layer Drivers	166
15-5-1	FSDev_NOR_AMD_1x08, FSDev_NOR_AMD_1x16	167
15-5-2	FSDev_NOR_Intel_1x16	167
15-5-3	FSDev_NOR_SST39	168
15-5-4	FSDev_NOR_STM25	168
15-5-5	FSDev_NOR_SST25	169
Chapter 16	RAM Disk Driver	170
16-1	Files and Directories	170
16-2	Using the RAM Disk Driver	171
Chapter 17	SD/MMC Drivers	174

17-1	Files and Directories	176
17-2	Using the SD/MMC CardMode Driver	177
17-2-1	SD/MMC CardMode Communication	180
17-2-2	SD/MMC CardMode Communication Debugging	182
17-2-3	SD/MMC CardMode BSP Overview	187
17-3	Using the SD/MMC SPI Driver	189
17-3-1	SD/MMC SPI Communication	193
17-3-2	SD/MMC SPI Communication Debugging	194
17-3-3	SD/MMC SPI BSP Overview	197
Appendix A	µC/FS API Reference Manual	198
A-1	General File System Functions	200
A-1-1	FS_DevDrvAdd()	201
A-1-2	FS_Init()	203
A-1-3	FS_VersionGet()	204
A-1-4	FS_WorkingDirGet()	205
A-1-5	FS_WorkingDirSet()	206
A-2	Posix API Functions	207
A-2-1	fs_asctime_r()	210
A-2-2	fs_chdir()	211
A-2-3	fs_clearerr()	212
A-2-4	fs_closedir()	213
A-2-5	fs_ctime_r()	214
A-2-6	fs_fclose()	215
A-2-7	fs_feof()	216
A-2-8	fs_ferror()	217
A-2-9	fs_fflush()	218
A-2-10	fs_fgetpos()	219
A-2-11	fs_flockfile()	220
A-2-12	fs_fopen()	221
A-2-13	fs_fread()	222
A-2-14	fs_fseek()	223
A-2-15	fs_fsetpos()	225
A-2-16	fs_ftell()	226
A-2-17	fs_ftruncate()	227
A-2-18	fs_ftrylockfile()	228
A-2-19	fs_funlockfile()	229
A-2-20	fs_fwrite()	230

A-2-21	fs_getcwd()	231
A-2-22	fs_localtime_r()	232
A-2-23	fs_mkdir()	233
A-2-24	fs_mktime()	234
A-2-25	fs_opendir()	235
A-2-26	fs_readdir_r()	236
A-2-27	fs_remove()	237
A-2-28	fs_rename()	239
A-2-29	fs_rewind()	241
A-2-30	fs_rmdir()	242
A-2-31	fs_setbuf()	244
A-2-32	fs_setvbuf()	245
A-3	Device Functions	247
A-3-1	FSDev_Close()	249
A-3-2	FSDev_GetDevName()	250
A-3-3	FSDev_GetDevCnt()	251
A-3-4	FSDev_GetDevCntMax()	252
A-3-5	FSDev_GetNbrPartitions()	253
A-3-6	FSDev_Open()	254
A-3-7	FSDev_PartitionAdd()	256
A-3-8	FSDev_PartitionFind()	257
A-3-9	FSDev_PartitionInit()	259
A-3-10	FSDev_Query()	261
A-3-11	FSDev_Rd()	262
A-3-12	FSDev_Refresh()	264
A-3-13	FSDev_Wr()	266
A-4	Directory Access Functions	267
A-4-1	FSDir_Close()	268
A-4-2	FSDir_IsOpen()	269
A-4-3	FSDir_Open()	270
A-4-4	FSDir_Rd()	272
A-5	Entry Access Functions	273
A-5-1	FSEntry_AttribSet()	274
A-5-2	FSEntry_Copy()	276
A-5-3	FSEntry_Create()	278
A-5-4	FSEntry_Del()	280
A-5-5	FSEntry_Query()	282
A-5-6	FSEntry_Rename()	284

A-5-7	FSEntry_TimeSet()	286
A-6	File Functions	288
A-6-1	FSFile_BufAssign()	290
A-6-2	FSFile_BufFlush()	292
A-6-3	FSFile_Close()	293
A-6-4	FSFile_ClrErr()	294
A-6-5	FSFile_IsEOF()	295
A-6-6	FSFile_IsErr()	296
A-6-7	FSFile_IsOpen()	297
A-6-8	FSFile_LockAccept()	299
A-6-9	FSFile_LockGet()	300
A-6-10	FSFile_LockSet()	301
A-6-11	FSFile_Open()	302
A-6-12	FSFile_PosGet()	305
A-6-13	FSFile_PosSet()	306
A-6-14	FSFile_Query()	308
A-6-15	FSFile_Rd()	309
A-6-16	FSFile_Truncate()	311
A-6-17	FSFile_Wr()	312
A-7	Volume Functions	314
A-7-1	FSVol_Close()	316
A-7-2	FSVol_Fmt()	317
A-7-3	FSVol_GetDfltVolName()	319
A-7-4	FSVol_GetVolCnt()	320
A-7-5	FSVol_GetVolCntMax()	321
A-7-6	FSVol_GetVolName()	322
A-7-7	FSVol_IsDflt()	323
A-7-8	FSVol_IsMounted()	324
A-7-9	FSVol_LabelGet()	325
A-7-10	FSVol_LabelSet()	327
A-7-11	FSVol_Open()	329
A-7-12	FSVol_Query()	331
A-7-13	FSVol_Rd()	332
A-7-14	FSVol_Wr()	334
A-8	Volume Cache Functions	335
A-8-1	FSVol_CacheAssign ()	336
A-8-2	FSVol_CacheInvalidate ()	338
A-8-3	FSVol_CacheFlush ()	339

A-9	NAND Driver Functions	340
A-9-1	FSDev_NAND_LowFmt()	341
A-9-2	FSDev_NAND_LowMount()	342
A-9-3	FSDev_NAND_LowUnmount()	343
A-9-4	FSDev_NAND_PhyRdSec()	344
A-9-5	FSDev_NAND_PhyWrSec()	346
A-9-6	FSDev_NAND_PhyEraseBlk()	348
A-10	NOR Driver Functions	350
A-10-1	FSDev_NOR_LowFmt()	352
A-10-2	FSDev_NOR_LowMount()	353
A-10-3	FSDev_NOR_LowUnmount()	354
A-10-4	FSDev_NOR_LowCompact()	355
A-10-5	FSDev_NOR_LowDefrag()	356
A-10-6	FSDev_NOR_PhyRd()	357
A-10-7	FSDev_NOR_PhyWr()	359
A-10-8	FSDev_NOR_PhyEraseBlk()	361
A-10-9	FSDev_NOR_PhyEraseChip()	363
A-11	SD/MMC Driver Functions	364
A-11-1	FSDev_SD_xxx_QuerySD()	365
A-11-2	FSDev_SD_xxx_RdCID()	367
A-11-3	FSDev_SD_xxx_RdCSD()	369
A-12	FAT System Driver Functions	370
A-12-1	FS_FAT_JournalOpen()	371
A-12-2	FS_FAT_JournalClose()	372
A-12-3	FS_FAT_JournalStart()	373
A-12-4	FS_FAT_JournalStop()	374
A-12-5	FS_FAT_VolChk()	375
Appendix B	µC/FS Error Codes	376
B-1	System Error Codes	376
B-2	Buffer Error Codes	376
B-3	Cache Error Codes	377
B-4	Device Error Codes	377
B-5	Device Driver Error Codes	378
B-6	Directory Error Codes	378
B-7	ECC Error Codes	378
B-8	Entry Error Codes	378
B-9	File Error Codes	379

B-10	Name Error Codes	379
B-11	Partition Error Codes	380
B-12	Pools Error Codes	380
B-13	File System Error Codes	380
B-14	Volume Error Codes	381
B-15	OS Layer Error Codes	382
Appendix C	µC/FS Porting Manual	383
C-1	Date/Time management	385
C-2	CPU Port	386
C-3	OS Kernel	386
C-4	Device Driver	394
C-4-1	NameGet()	396
C-4-2	Init()	397
C-4-3	Open()	398
C-4-4	Close()	400
C-4-5	Rd()	401
C-4-6	Wr()	402
C-4-7	Query()	404
C-4-8	IO_Ctrl()	406
C-5	IDE/CF Device BSP	408
C-5-1	FSDev_IDE_BSP_Open()	410
C-5-2	FSDev_IDE_BSP_Close()	411
C-5-3	FSDev_IDE_BSP_Lock() / FSDev_IDE_BSP_Unlock()	412
C-5-4	FSDev_IDE_BSP_Reset()	413
C-5-5	FSDev_IDE_BSP_RegRd()	414
C-5-6	FSDev_IDE_BSP_RegWr()	415
C-5-7	FSDev_IDE_BSP_CmdWr()	416
C-5-8	FSDev_IDE_BSP_DataRd()	417
C-5-9	FSDev_IDE_BSP_DataWr()	418
C-5-10	FSDev_IDE_BSP_DMA_Start()	419
C-5-11	FSDev_IDE_BSP_DMA_End()	420
C-5-12	FSDev_IDE_BSP_GetDrvNbr()	422
C-5-13	FSDev_IDE_BSP_GetModesSupported()	423
C-5-14	FSDev_IDE_BSP_SetMode()	424
C-5-15	FSDev_IDE_BSP_Dly400_ns()	425
C-6	NAND Flash Physical-Layer Driver	426
C-6-1	Open()	429

C-6-2 Close() 431
C-6-3 RdPage() 432
C-6-4 RdSpare() 434
C-6-5 WrPage() 435
C-6-6 WrSpare() 436
C-6-7 CopyBack() 437
C-6-8 EraseBlk() 438
C-6-9 IO_Ctrl() 439
C-7 NAND Flash BSP 440
C-7-1 FSDev_NAND_BSP_Open() 441
C-7-2 FSDev_NAND_BSP_Close() 442
C-7-3 FSDev_NAND_BSP_ChipSelEn() 443
C-7-4 FSDev_NAND_BSP_ChipSelDis() 444
C-7-5 FSDev_NAND_BSP_RdData() 445
C-7-6 FSDev_NAND_BSP_WrAddr() 446
C-7-7 FSDev_NAND_BSP_WrCmd() 447
C-7-8 FSDev_NAND_BSP_WrData() 448
C-7-9 FSDev_NAND_BSP_WaitWhileBusy() 449
C-8 NAND Flash SPI BSP 450
C-9 NOR Flash Physical-Layer Driver 450
C-9-1 Open() 453
C-9-2 Close() 454
C-9-3 Rd() 455
C-9-4 Wr() 456
C-9-5 EraseBlk() 457
C-9-6 IO_Ctrl() 458
C-10 NOR Flash BSP 459
C-10-1 FSDev_NOR_BSP_Open() 460
C-10-2 FSDev_NOR_BSP_Close() 461
C-10-3 FSDev_NOR_BSP_Rd_XX() 462
C-10-4 FSDev_NOR_BSP_RdWord_XX() 463
C-10-5 FSDev_NOR_BSP_WrWord_XX() 464
C-10-6 FSDev_NOR_BSP_WaitWhileBusy() 465
C-11 NOR Flash SPI BSP 466
C-12 SD/MMC Cardmode BSP 467
C-12-1 FSDev_SD_Card_BSP_Open() 470
C-12-2 FSDev_SD_Card_BSP_Lock() 471
C-12-3 FSDev_SD_Card_BSP_CmdStart() 472

C-12-4	FSDev_SD_Card_BSP_CmdWaitEnd()	477
C-12-5	FSDev_SD_Card_BSP_CmdDataRd()	481
C-12-6	FSDev_SD_Card_BSP_CmdDataWr()	484
C-12-7	FSDev_SD_Card_BSP_GetBlkCntMax()	487
C-12-8	FSDev_SD_Card_BSP_GetBusWidthMax()	488
C-12-9	FSDev_SD_Card_BSP_SetBusWidth()	489
C-12-10	FSDev_SD_Card_BSP_SetClkFreq()	491
C-12-11	FSDev_SD_Card_BSP_SetTimeoutData()	492
C-12-12	FSDev_SD_Card_BSP_SetTimeoutResp()	493
C-13	SD/MMC SPI mode BSP	493
C-14	SPI BSP	494
C-14-1	Open()	499
C-14-2	Close()	501
C-14-3	Lock() / Unlock()	502
C-14-4	Rd()	503
C-14-5	Wr()	504
C-14-6	ChipSelEn() /ChipSelDis()	505
C-14-7	SetClkFreq()	506
Appendix D	μC/FS Types and Structures	507
D-1	FS_CFG	508
D-2	FS_DEV_INFO	510
D-3	FS_DEV_NAND_CFG	511
D-4	FS_DEV_NOR_CFG	513
D-5	FS_DEV_RAM_CFG	516
D-6	FS_DIR_ENTRY (struct fs_dirent)	517
D-7	FS_ENTRY_INFO	518
D-8	FS_FAT_SYS_CFG	520
D-9	FS_PARTITION_ENTRY	522
D-10	FS_VOL_INFO	523
Appendix E	μC/FS Configuration	525
E-1	File System Configuration	526
E-2	Feature Inclusion Configuration	527
E-3	Name Restriction Configuration	530
E-4	Debug Configuration	531
E-5	Argument Checking Configuration	531

E-6	File System Counter Configuration	532
E-7	Fat Configuration	532
E-8	SD/MMC SPI Configuration	533
E-9	Trace Configuration	534
Appendix F	Shell Commands	535
F-1	Files and Directories	536
F-2	Using the Shell Commands	537
F-3	Commands	540
F-3-1	fs_cat	541
F-3-2	fs_cd	542
F-3-3	fs_cp	544
F-3-4	fs_date	545
F-3-5	fs_df	546
F-3-6	fs_ls	547
F-3-7	fs_mkdir	548
F-3-8	fs_mkfs	549
F-3-9	fs_mount	550
F-3-10	fs_mv	551
F-3-11	fs_od	552
F-3-12	fs_pwd	553
F-3-13	fs_rm	554
F-3-14	fs_rmdir	555
F-3-15	fs_touch	556
F-3-16	fs_umount	557
F-3-17	fs_wc	558
F-4	Configuration	559
Appendix G	Bibliography	561
Appendix H	µC/FS Licensing Policy	563
H-1	µC/FS Licensing	563
H-1-1	µC/FS Source Code	563
H-1-2	µC/FS Maintenance Renewal	564
H-1-3	µC/FS Source Code Updates	564
H-1-4	µC/FS Support	564

Chapter

1

Introduction

Files and directories are common abstractions, which we encounter daily when sending an e-mail attachment, downloading a new application or archiving old information. Those same abstractions may be leveraged in an embedded system for similar tasks or for unique ones. A device may serve web pages, play or record media (images, video or music) or log data. The file system software which performs such actions must meet the general expectations of an embedded environment—a limited code footprint, for instance—which still delivering good performance.

1-1 WHAT IS A FILE SYSTEM?

A file system is a collection of files and directories; since directories are containers of files, a hierarchical organization results. A PC operating system such as Windows or Linux presents its file systems through a visual interface (e.g, "Windows Explorer"), with a tree-like structure of entries that can be moved, renamed or deleted with menus or actions like "dragging and dropping". Alternatively, a headless system like DOS (or any other command line) integrates utilities to accomplish the same operations.

Above, we stated that a system "presents its file systems"—file systems plural—because each drive is a separate file system, a separate collection of files. Each of these is anchored by some unique drive letter (Windows) or mount point (Linux) within the larger context of a "virtual" file system wherein every entry has a unique identifier. (Within the "everything is a file" mentality of Linux, this is taken further, but that is beyond this discussion.) Being separate, each file system may have a different format—one may be FAT, the next NTFS—and will be located on different physical devices or on separate partitions of the same device.

If files are to be read from a volume, file system software is required, with three basic elements. First, a device driver must be able to read and write to the device. Next, a file system driver must be able to parse the device's on-disk structures to read the names, properties and data of files and to format those structures to modify existing entries and create new ones. Finally, an application-level interface must provide for the exigencies of file and directory access.

1-2 µC/FS

µC/FS is a compact, reliable, high-performance file system. It offers full-featured file and directory access with flexible device and volume management including support for partitions.

Source Code: µC/FS is provided in ANSI-C source to licensees. The source code is written to an exacting coding standard that emphasizes cleanness and readability. Moreover, extensive comments pepper the code to elucidate its logic and describe global variables and functions. Where appropriate, the code directly references standards and supporting documents.

Device Drivers: Device drivers are available for most common media including SD/MMC cards, NAND flash, NOR flash and IDE/CF. Each of these is written with a clear, 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 µC/FS can easily be ported to a new medium.

Devices and Volumes: Multiple media can be accessed simultaneously, including multiple instances of the same type of medium (since all drivers are re-entrant). DOS partitions are supported, so more than one volume can be located on a device. In addition, the logical device driver allows a single volume to span several (typically identical) devices, such as a bank of flash chips.

FAT: All standard FAT variants and features are supported including FAT12/FAT16/FAT32 and long file names, which encompasses Unicode file names. Files can be up to 4-GB and volumes up to 8-TB (the standard maximum). An optional journaling module provides total power fail-safety to the FAT system driver.

Application Programming Interface (API): µC/FS provides two APIs for file and directory access. A standard POSIX-compatible API is provided, including functions like fs_fwrite(), fs_fread() and fs_fsetpos() that have the same arguments and return values as the POSIX functions fwrite(), fread() and fsetpos(). Another API with parallel argument placement and meaningful return error codes is provided as an alternate, with functions like FSFile_Wr(), FSFile_Rd() and FSFile_PosSet().

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 sufficient RAM, these features can be enabled in order to gain better performance.

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

RTOS: μC/FS does not assume the presence of a RTOS kernel. However, if you are using a 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 a RTOS, this port layer may consist of empty functions.

1-3 TYPICAL USAGES

Applications have sundry reasons for non-volatile storage. A subset require (or benefit from) organizing data into named files within a directory hierarchy on a volume—basically, from having a file system. Perhaps the most obvious expose the structure of information to the user, like products that store images, video or music that are transferred to or from a PC. A web interface poses a similar opportunity, since the URLs of pages and images fetched by the remote browser would resolve neatly to locations on a volume.

Another typical use is data logging. A primary purpose of a device may be to collect data from its environment for later retrieval. If the information must persist across device reset events or will exceed the capacity of its RAM, some non-volatile memory is necessary. The benefit of a file system is the ability to organize that information logically, with a fitting directory structure, through a familiar API.

A file system can also store programs. In a simple embedded CPU, the program is stored at a fixed location in a non-volatile memory (usually flash). If an application must support firmware updates, a file system may be a more convenient place, since the software handles the details of storing the program. The boot-loader, of course, would need to be able to load the application, but since that requires only read-only access, no imposing program is required. The ROM boot-loaders in some CPUs can check the root directory of a SD card for a binary in addition to the more usual locations such as external NAND or NOR flash.

1-4 WHY FAT?

File Allocation Table (FAT) is a simple file system, widely supported across major OSs. While it has been supplanted as the format of hard drives in Windows PCs, removable media still use FAT because of its wide support. That is suitable for embedded systems, which would often be challenged to muster the resources for the modern file systems developed principally for large fixed disks.

µC/FS supports FAT because of the interoperability requirements of removable media, allowing that a storage medium be removed from an embedded device and connected to a PC. All variants and extensions are supported to specification.

A notorious weakness of FAT (exacerbated by early Windows system drivers) is its non-fail safe architecture. Certain operations leave the file system in an inconsistent state, albeit briefly, which may corrupt the disk or force a disk check upon unexpected power failure. µC/FS minimizes the problem by ordering modifications wisely. The problem is completely solved in an optional journaling module which logs information about pending changes so those can be resumed on start-up after a power failure.

1-5 CHAPTER CONTENTS

Figure 1-1 shows the layout and flow of the book. This diagram should be useful to understand the relationship between chapters. The first (leftmost) column lists chapters that should be read in order to understand µC/FS's structure. The chapters in the second column give greater detail about the application of µC/FS. Each of the chapters in the third column examines a storage technology and its device driver. Finally, the fourth column lists the appendices, the topmost being the µC/FS reference, configuration and porting manuals. Reference these sections regularly when designing a product using µC/FS.

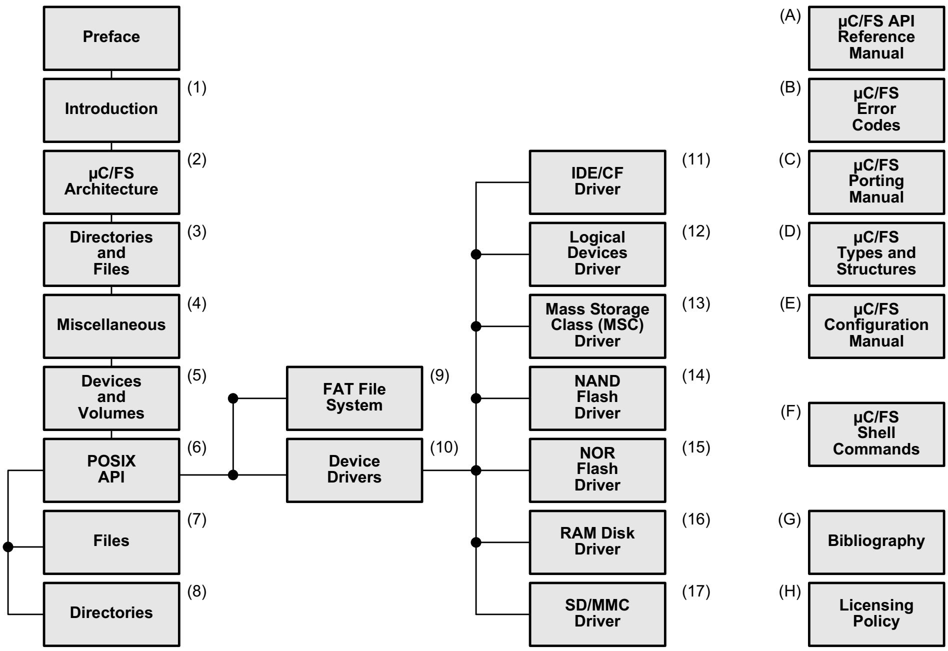


Figure 1-1 **µC/FS Book Layout**

Chapter 1, Introduction. This chapter.

Chapter 2, µC/FS Architecture. This chapter contains a simplified block diagram of the various different µC/FS modules and their relationships. The relationships are then explained.

Chapter 3, Directories and Files. This chapter explains the directory structure and files needed to build a µC/FS-based application. Learn about the files that are needed, where they should be placed, which module does what, and more.

Chapter 4, Miscellaneous. In this chapter, you will learn the nomenclature used in µC/FS to access files and folders and the ressources needed to use µC/FS in your application.

Chapter 5, Devices and Volumes. Every file and directory accessed with µC/FS is a constituent of a volume (a collection of files and directories) on a device (a physical or logical sector-addressed entity). This chapter explains how devices and volumes are managed.

Chapter 6, POSIX API. The best-known API for accessing and managing files and directories is specified within the POSIX standard (IEEE Std 1003.1), which is based in part in the ISO C standard (ISO/IEC 9899). This chapter explains how to use this API and examines some of its pitfalls and shortcomings.

Chapter 7, Files. µC/FS complements the POSIX API with its own file access API. This chapter explains this API.

Chapter 8, Directories. µC/FS complements the POSIX API with its own directory access API. This chapter explains this API.

Chapter 9, FAT File System. This chapter details the low-level architecture of the FAT file system. Though the API of µC/FS is file system agnostic, the file system type does affect performance, reliability and security, as explained here as well.

Chapter 10, Device Drivers. All hardware accesses are eventually performed by a device driver. This chapter describes the drivers available with µC/FS and broadly profiles supported media types in terms of cost, performance and complexity.

Chapter 11, IDE Devices. The IDE driver supports compact flash (CF) cards and ATA IDE hard drives.

Chapter 12, Logical Devices Driver. This feature is not available yet.

Chapter 13, Mass Storage Class (MSC) Driver. The now-common USB drive implements the Mass Storage Class (MSC) protocol, and a CPU with a USB host interface can access these devices with appropriate software. The MSC driver, discussed in this chapter, with µC/USB-Host is just such appropriate software.

Chapter 14, NAND Flash. NAND flash is the first category of flash media. Write speeds are fast (compared to NOR flash), at the expense of slower read speeds and complexities such as bit-errors and page program limitations. This chapter describes the functions of these devices and the architecture of the supporting driver.

Chapter 15, NOR Flash. NOR flash is the second category of flash media. They suffer slow write speeds, balanced with blazingly-fast read speeds. Importantly, they are not plagued by the complications of NAND flash, which simplifies interfacing with them. This chapter describes the function of these devices and the architecture of the supporting driver.

Chapter 16, RAM Disk. This chapter demonstrates the use of the simplest storage medium, the RAM disk.

Chapter 17, SD/MMC Devices. SD and MMC cards are flash-based removable storage devices commonly used in consumer electronics. For embedded CPUs, a SD/MMC card is an appealing medium because of its simple and widely-supported physical interfaces (one choice is SPI). This chapter describes the interface and function of these devices.

Appendix A, µC/FS API Reference Manual. The reference manual describes every API function. The arguments and return value of each function are given, supplemented by notes about its use and an example code listing.

Appendix B, µC/FS Error Codes. This appendix provides a brief explanation of µC/FS error codes defined in `fs_err.h`.

Appendix C, µC/FS Porting Manual. The portability of µC/FS relies upon ports to interface between its modules and the platform or environment. Most of the ports constitute the board support package (BSP), which is interposed between the file system suite (or driver) and hardware. The OS port adapts the software to a particularly OS kernel. The porting manual describes each port function.

Appendix D, µC/FS Types and Structures. This appendix provides a reference to the µC/FS types and structures.

Appendix E, µC/FS API Configuration Manual. µC/FS is configured via defines in a single configuration file, `fs_cfg.h.` The configuration manual specifies each define and the meaning of possible values.

Appendix F, µC/FS Shell Commands. A familiar method of accessing a file system, at least to engineers and computer scientists, is the command line. In an embedded system, a UART is a port over which commands can be executed easily, even for debug purposes. A set of shell commands have been developed for µC/FS that mirror the syntax of UNIX utilities, as described in this chapter.

Appendix G, Bibliographhy.

Appendix H, Licensing Policy.

Chapter

2

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 2-1 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_`**'.

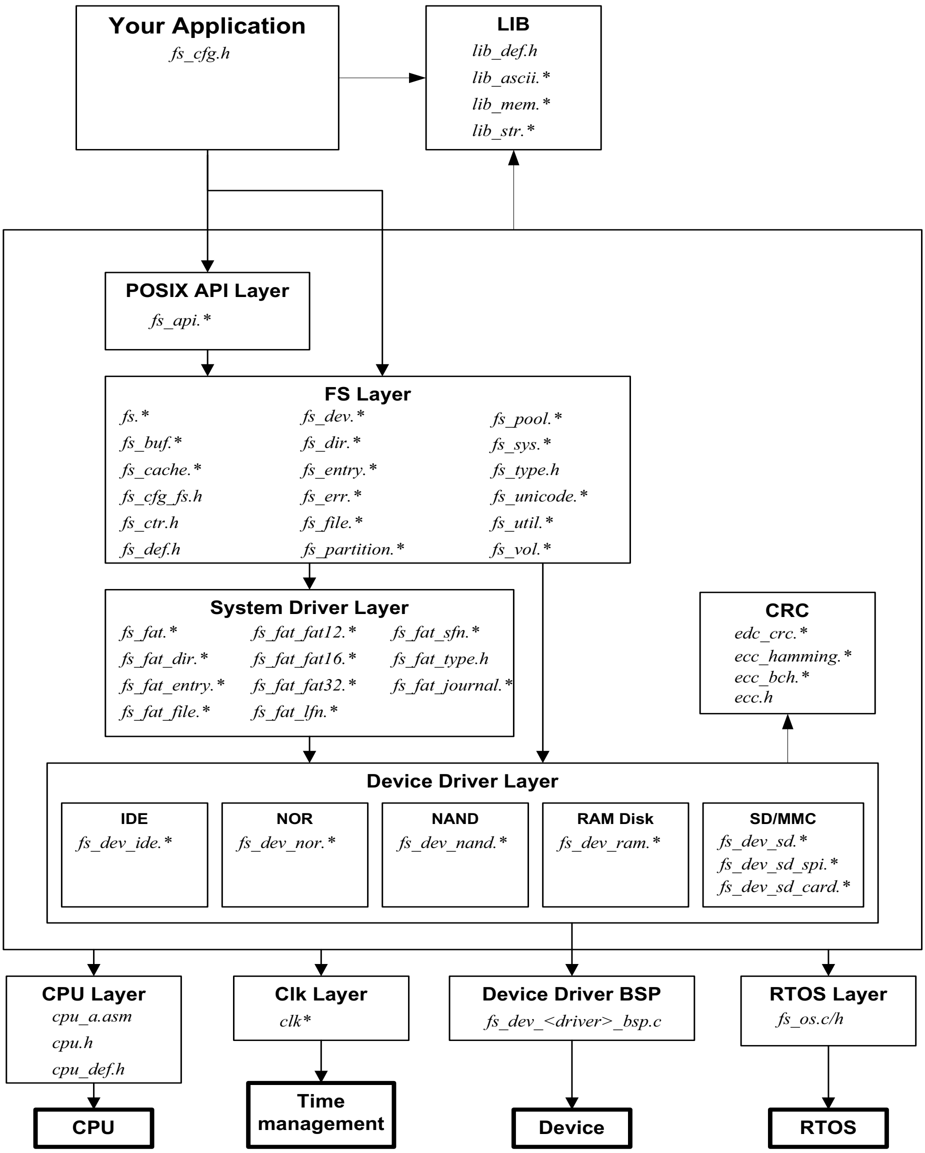


Figure 2-1 **µC/FS architecture.**

2-1 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.

2-1-1 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.

2-1-2 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.

2-1-3 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.

2-1-4 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:

- Devices (`fs_dev.*`)

- Directories (`fs_dir.*`)
- Entries (`fs_entry.*`)
- Files (`fs_file.*`)
- Partitions (`fs_partition.*`)
- Volumes (`fs_vol.*`)

2 Support files:

- Buffer management (`fs_buf.*`)
- Cache management (`fs_cache.*`)
- Counter management (`fs_ctr.h`)
- Pool management (`fs_pool.*`)
- File system driver (`fs_sys.*`)
- Unicode encoding support (`fs_unicode.*`)
- Utility functions (`fs_util.*`)

3 Miscellaneous header files:

- Master µC/FS header file (`fs.h`)
- Error codes (`fs_err.h`)
- Miscellaneous data types (`fs_type.h`)
- Miscellaneous definitions (`fs_def.h`)
- Configuration definitions (`fs_cfg_fs.h`)

2-1-5 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.

2-1-6 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.

2-1-7 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.

2-1-8 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 (which contains just empty functions), a µC/OS-II and a µC/OS-III 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.

Chapter

3

Directories and Files

µC/FS is fairly easy to use once you understand which source files are needed to make up a µC/FS-based application. This chapter will discuss the modules available for µC/FS and how everything fits together.

Figure 1-01 shows the µC/FS architecture and its relationship with the hardware. Memory devices may include actual media both removable (SD/MMC, CF cards) and fixed (NAND flash, NOR flash) as well as any controllers for such devices. Of course, your hardware would most likely contain other devices such as UARTs (Universal Asynchronous Receiver Transmitters), ADCs (Analog to Digital Converters) and Ethernet controller(s). Moreover, your application may include other middleware components like an OS kernel, networking (TCP/IP) stack or USB stack that may integrate with µC/FS.

A Windows™-based development platform is assumed. The directories and files make references to typical Windows-type directory structures. However, since µC/FS is available in source form then it can certainly be used on Unix, Linux or other development platforms. This, of course, assumes that you are a valid µC/FS licensee in order to obtain the source code.

The names of the files are shown in upper case to make them ‘stand out’. The file names, however, are actually lower case.

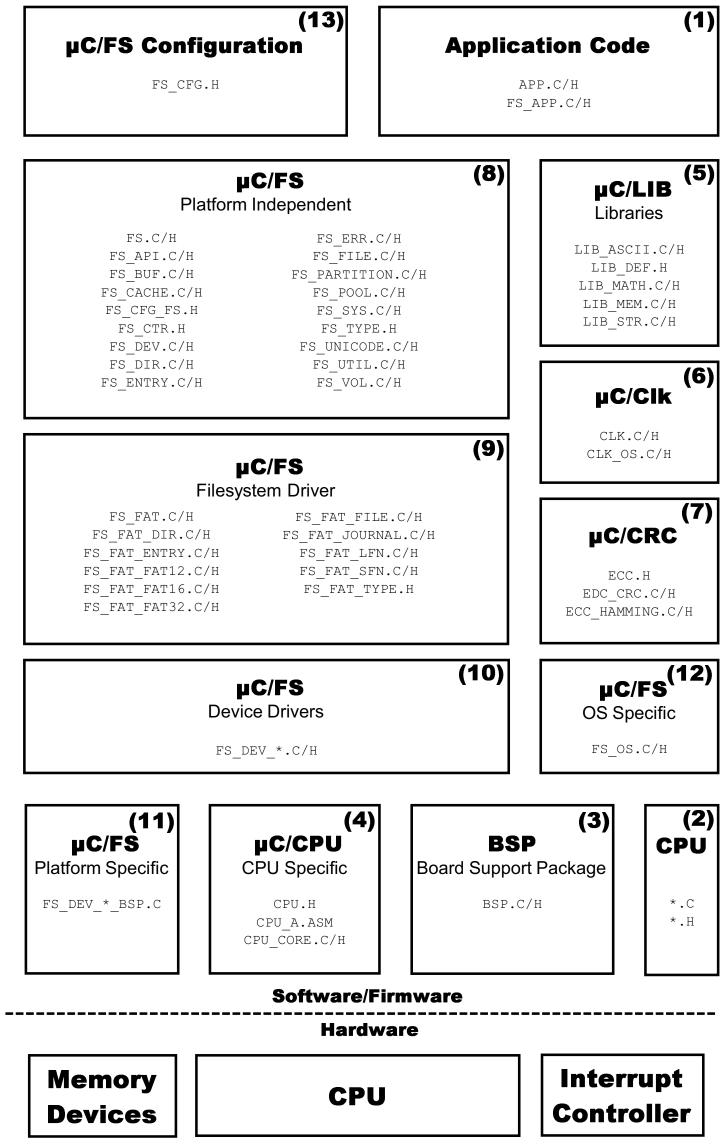


Figure 3-1 **µC/FS Architecture**

F3-1(1) The application code consist of project or product files. For convenience, we simply called these `app.c` and `app.h` but your application can contain any number of files and they do not have to be called `app.*`. The application code is typically where you would find `main()`.

F3-1(2) Quite often, semiconductor manufacturers provide library functions in source form for accessing the peripherals on their CPU (Central Processing Unit) or MCU (Micro Controller Unit). These libraries are quite useful and often save valuable time. Since there is no naming convention for these files, *.`c` and *.`h` are assumed.

F3-1(3) The Board Support Package (BSP) is code that you would typically write to interface to peripherals on your target board. For example you can have code to turn on and off LEDs (light emitting diodes), functions to turn on and off relays, and code to read switches and temperature sensors.

F3-1(4) At Micriμm, we like to encapsulate CPU functionality. These files define functions to disable and enable interrupts, data types (e.g., `CPU_INT08U`, `CPU_FP32`) independent of the CPU and compiler and many more functions.

F3-1(5) μC/LIB consists of a group of source files to provide common functions for memory copy, string manipulation and character mapping. Some of the functions replace stdlib functions provided by the compiler. These are provided to ensure that they are fully portable from application to application and (most importantly) from compiler to compiler.

F3-1(6) μC/Clk is an independant clock/calendar management module, with source code for easily managing date and time in a product. μC/FS uses the date and time information from μC/Clk to update files and directories with the proper creation/modification/access time.

F3-1(7) μC/CRC is a stand-alone module for calculating checksums and error correction codes. This module is used by some of μC/FS device drivers.

F3-1(8) This is the μC/FS platform-independent code, free of dependencies on CPU and memory device. This code is written in highly-portable ANSI C code. This code is only available to μC/FS licensees.

F3-1(9) This is the µC/FS system driver for FAT file systems. This code is only available to µC/FS licensees.

F3-1(10) This is the collection of device drivers for µC/FS. Each driver supports a certain device type, such as SD/MMC cards, NAND flash or NOR flash. Drivers are only available to µC/FS licensees.

F3-1(11) This is the µC/FS code that is adapted to a specific platform. It consists of small code modules written for specific drivers called ports that must be adapted to the memory device controllers or peripherals integrated into or attached to the CPU. The requirements for these ports are described in Appendix C, Porting Manual.

F3-1(12) 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.

F3-1(13) This µC/FS configuration file defines which µC/FS features (`fs_cfg.h`) are included in the application.

3-1 APPLICATION CODE

When Micriµm provides you with example projects, we typically place those in a directory structure as shown below. Of course, you can use whatever directory structure suits your project/product.

```
\Micrium
    \Software
        \EvalBoards
            \<manufacturer>
                \<board name>
                    \<compiler>
                        \<project name>
                            \*.*
```

\Micrium

This is where we place all software components and projects provided by Micriμm. This directory generally starts from the root directory of your computer.

\Software

This sub-directory contains all the software components and projects.

\EvalBoards

This sub-directory contains all the projects related to the evaluation boards supported by Micriμm.

\<manufacturer>

Is the name of the manufacturer of the evaluation board. The '<' and '>' are not part of the actual name.

\<board name>

This is the name of the evaluation board. A board from Micriμm will typically be called uC-Eval-xxxx where 'xxxx' will represent the CPU or MCU used on the evaluation board. The '<' and '>' are not part of the actual name.

\<compiler>

This is the name of the compiler or compiler manufacturer used to build the code for the evaluation board. The '<' and '>' are not part of the actual name.

\<project name>

This is the name of the project that will be demonstrated. For example a simple μC/FS project might have a project name of 'FS-Ex1'. The '-Ex1' represents a project containing only μC/FS. A project name of FS-Probe-Ex1 would represent a project containing μC/FS as well as μC/Probe. The '<' and '>' are not part of the actual name.

.

These are the source files for the project/product. You are certainly welcomed to call the main files APP*.* for your own projects but you don't have to. This directory also contains the configuration file `FS_CFG.H` and other files as needed by the project.

3-2 CPU

As shown below is the directory where we place semiconductor manufacturer peripheral interface source files. Of course, you can use whatever directory structure suits your project/product.

```
\Micrium
    \Software
        \CPU
            \<manufacturer>
                \<architecture>
                    \*.*
```

`\Micrium`
This is where we place all software components and projects provided by Micriμm.

`\Software`
This sub-directory contains all the software components and projects.

`\CPU`
This sub-directory is always called CPU.

`\<manufacturer>`
Is the name of the semiconductor manufacturer who provided the peripheral library. The ‘<’ and ‘>’ are not part of the actual name.

`\<architecture>`
This is the name of the specific library and is generally associated with a CPU name or an architecture.

`\*.*`
These are the library source files. The names of the files are determined by the semiconductor manufacturer.

3-3 BOARD SUPPORT PACKAGE (BSP)

The BSP is generally found with the evaluation or target board because the BSP is specific to that board. In fact, if well written, the BSP should be used for multiple projects.

```
\Micrium
    \Software
        \EvalBoards
            \<manufacturer>
                \<board name>
                    \<compiler>
                        \BSP
                            \*.*
```

\Micrium
This is where we place all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

\EvalBoards
This sub-directory contains all the projects related to evaluation boards.

\<manufacturer>
Is the name of the manufacturer of the evaluation board. The '<' and '>' are not part of the actual name.

\<board name>
This is the name of the evaluation board. A board from Micriµm will typically be called uC Eval xxxx where 'xxxx' will be the name of the CPU or MCU used on the evaluation board. The '<' and '>' are not part of the actual name.

\<compiler>
This is the name of the compiler or compiler manufacturer used to build the code for the evaluation board. The '<' and '>' are not part of the actual name.

\BSP
This directory is always called BSP.

.

These are the source files of the BSP. Typically all the file names start with BSP_ but they don't have to. It's thus typical to find **bsp.c** and **bsp.h** in this directory. Again, the BSP code should contain functions such as LED control functions, initialization of timers, interface to Ethernet controllers and more.

3-4 µC/CPU, CPU SPECIFIC SOURCE CODE

µC/CPU consists of files that encapsulate common CPU-specific functionality as well as CPU- and compiler-specific data types.

```
\Micrium
    \Software
        \uC-CPU
            \cpu_core.c
            \cpu_core.h
            \cpu_def.h
            \Cfg\Template
                \cpu_cfg.h
            \<architecture>
                \<compiler>
                    \cpu.h
                    \cpu_a.asm
                    \cpu_c.c
```

\Micrium

This directory contains all software components and projects provided by Micriµm.

\Software

This sub-directory contains all the software components and projects.

\uC-CPU

This is the main µC/CPU directory.

cpu_core.c contains C code that is common to all CPU architectures. Specifically, this file contains functions to measure the interrupt disable time of the CPU_CRITICAL_ENTER() and CPU_CRITICAL_EXIT() macros, a function that emulates a count leading zeros instruction and a few other functions.

cpu_core.h contains the function prototypes of the functions provided in cpu_core.c as well as allocation of the variables used by this module to measure interrupt disable time.

cpu_def.h contains miscellaneous #define constants used by the µC/CPU module.

\Cfg\Template
This directory contains a configuration template file (cpu_cfg.h) that you will need to copy to your application directory in order to configure the µC/CPU module based on your application requirements.

cpu_cfg.h determines whether you will enable measurement of the interrupt disable time, whether your CPU implements a count leading zeros instruction in assembly language or whether it will need to be emulated in C and more.

\<architecture>
This is the name of the CPU architecture for which µC/CPU was ported to. The '<' and '>' are not part of the actual name.

\<compiler>
This is the name of the compiler or compiler manufacturer used to build the code for the µC/CPU port. The '<' and '>' are not part of the actual name.

The files in this directory contain the µC/CPU port.

cpu.h contains type definitions to make µC/FS and other modules independent of the CPU and compiler word sizes. Specifically, you will find the declaration of the CPU_INT16U, CPU_INT32U, CPU_FP32 and many other data types. Also, this file specifies whether the CPU is a big- or little-endian machine and contains function prototypes for functions that are specific to the CPU architecture and more.

cpu_a.asm contains the assembly language functions to implement the code to disable and enable CPU interrupts, count leading zeros (if the CPU supports that instruction) and other CPU specific functions that can only be written in assembly language. This file could also contain code to enable caches, setup MPUs and MMU and more. The functions provided in this file are accessible from C.

cpu_c.c contains C code of functions that are specific to the specific CPU architecture but written in C for portability. As a general rule, if a function can be written in C then it should, unless there are significant performance benefits by writing it in assembly language.

3-5 µC/LIB, PORTABLE LIBRARY FUNCTIONS

µC/LIB consists of library functions that are meant to be highly portable and not tied to any specific compiler. This was done to facilitate third party certification of Micriµm products.

```
\Micrium
    \Software
        \uC-LIB
            \lib_ascii.c
            \lib_ascii.h
            \lib_def.h
            \lib_math.c
            \lib_math.h
            \lib_mem.c
            \lib_mem.h
            \lib_str.c
            \lib_str.h
            \Cfg\Template
                \lib_cfg.h
            \Ports
                \<architecture>
                    \<compiler>
                        \lib_mem_a.asm
```

\Micrium

This directory contains all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

\uC-LIB
This is the main µC/LIB directory.

\Cfg\Template
This directory contains a configuration template file (`lib_cfg.h`) that must be copied to the application directory to configure the µC/LIB module based on application requirements.

> **lib_cfg.h** determines whether to enable assembly-language optimization (assuming there is an assembly-language file for the processor, i.e. `lib_mem_a.asm`) and a few other #defines.

3-6 µC/CLK, TIME/CALENDAR MANAGEMENT

µC/Clk consists of functions that are meant to centralize time management in one independant module. This way, the same time info can be easily shared across all Micrium products.

```
\Micrium
    \Software
        \uC-Clk
            \Cfg
                \Template
                    \clk_cfg.h
            \OS
                \<rtos_name>
                    \clk_os.c
            \Source
                \clk.c
                \clk.h
```

\Micrium
This directory contains all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

\uC-Clk
This is the main µC/Clk directory.

\Cfg\Template
This directory contains a configuration template file (`clk_cfg.h`) that must be copied to the application directory to configure the µC/Clk module based on application requirements.

> **`clk_cfg.h`** determines whether clock will be managed by the RTOS or in your application. A few other #defines are used to enable/disable some features of µC/Clk and to configure some parameteres, like the clock frequency.

\OS
This is the main OS directory.

\<rtos_name>
This is the directory that contains the file to perform RTOS abstraction. Note that the file for the selected RTOS abstraction layer must always be named **`clk_os.c`**.

µC/Clk has been tested with µC/OS-II, µC/OS-III and the RTOS layer files for these RTOS are found in the following directories:

`\Micrium\Software\uC-Clk\OS\uCOS-II\clk_os.c`

`\Micrium\Software\uC-Clk\OS\uCOS-III\clk_os.c`

\Source
This directory contains the CPU-independant source code for µC/Clk. All file in this directory should be included in the build (assuming the presence of the source code). Features that are not required will be compiled out based on the value of #define constants in `clk_cfg.h`.

3-7 µC/CRC, CHECKSUMS AND ERROR CORRECTION CODES

µC/CRC consists of functions to compute different error detection and correction codes. The functions are speed-optimized to avoid the important impact on performances that these CPU-intensive calcutions may present.

```
\Micrium
    \Software
        \uC-CRC
            \Cfg
                \Template
                    \crc_cfg.h
            \Ports
                \<architecture>
                    \<compiler>
                        \ecc_bch_4bit_a.asm
                        \ecc_bch_8bit_a.asm
                        \ecc_hamming_a.asm
                        \edc_crc_a.asm
            \Source
                \edc_crc.h
                \edc_crc.c
                \ecc_hamming.h
                \ecc_hamming.c
                \ecc_bch_8bit.h
                \ecc_bch_8bit.c
                \ecc_bch_4bit.h
                \ecc_bch_4bit.c
                \ecc_bch.h
                \ecc_bch.c
                \ecc.h
```

\Micrium
This directory contains all software components and projects provided by Micriμm.

\Software
This sub-directory contains all the software components and projects.

\uC-CRC
This is the main µC/CRC directory.

\Cfg\Template
This directory contains a configuration template file (**crc_cfg.h**) that must be copied to the application directory to configure the µC/CRC module based on application requirements.

> **crc_cfg.h** determines whether to enable assembly-language optimization (assuming there is an assembly-language file for the processor) and a few other #defines.

\<architecture>
The name of the CPU architecture that µC/CRC was ported to. The ‘<’ and ‘>’ are not part of the actual name.

\<compiler>
The name of the compiler or compiler manufacturer used to build code for the µC/CRC port. The ‘<’ and ‘>’ are not part of the actual name.

> **ecc_bch_4bit_a.asm** contains the assembly language functions to optimize the calculation speed of 4-bit correction BCH (Bos, Ray-Chaudhuri, Hocquenghem) code.
>
> **ecc_bch_8bit_a.asm** contains the assembly language functions to optimize the calculation speed of 8-bit correction BCH (Bos, Ray-Chaudhuri, Hocquenghem) code.
>
> **ecc_hamming_a.asm** contains the assembly language functions to optimize the calculation speed of Hamming code.
>
> **edc_crc_a.asm** contains the assembly language functions to optimize the calculation speed of CRC (cyclic redundancy checks).

\Source
This is the directory that contains all the CPU independent source code files. of µC/CRC.

3-8 µC/FS PLATFORM-INDEPENDENT SOURCE CODE

The files in these directories are available to µC/FS licensees (see Appendix H, Licensing Policy).

```
\Micrium
    \Software
        \uC-FS
               \APP\Template
                \fs_app.c
                \fs_app.h
                     \Cfg\Template
                \fs_cfg.h
            \OS\Template
                \fs_os.c
                \fs_os.h
            \Source
                \fs_c
                \fs.h
                \fs_api.c
                \fs_api.h
                \fs_buf.c
                \fs_buf.h
                \fs_cache.c
                \fs_cache.h
                \fs_cfg_fs.h
                \fs_ctr.h
                \fs_def.h
                \fs_dev.c
                \fs_dev.h
                \fs_dir.c
                \fs_dir.h
                \fs_entry.c
                \fs_entry.h
                \fs_err.c
                \fs_err.h
                \fs_file.c
                \fs_file.h
```

\fs_partition.c
\fs_partition.h
\fs_pool.c
\fs_pool.h
\fs_sys.c
\fs_sys.h
\fs_type.h
\fs_unicode.c
\fs_unicode.h
\fs_util.c
\fs_util.h
\fs_vol.c
\fs_vol.h

\Micrium
This is where we place all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

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

\APP\Template
This directory contains a template of the code for initializing the file system.

\Cfg\Template
This directory contains a configuration template file (lib_cfg.h) that is required to be copied to the application directory to configure the µC/FS module based on application requirements.

fs_cfg.h specifies which features of µC/FS you want in your application. If µC/FS is provided in linkable object code format then this file will be provided to show you what features are available in the object file. See Appendix B, µC/FS Configuration Manual.

\Source
This directory contains the platform-independent source code for µC/FS. All the files in this directory should be included in your build (assuming you have the source code). Features that you don't want will be compiled out based on the value of #define constants in `fs_cfg.h`.

fs.c/h contain core functionality for µC/FS including `FS_Init()` (called to initialize µC/FS) and `FS_WorkingDirSet()`/`FS_WorkingDirGet()` (used to get and set the working directory). **fs.h** is the ONLY core header file that should be #included by the application.

fs_api.c/h contains the code for the POSIX-compatible API. See Chapter x, API for details about the POSIX-compatible API.

fs_buf.c/h contains the code for the buffer management (used internally by µC/FS).

fs_dev.c/h contains code for device management. See Chapter x, Devices for details about devices.

fs_dir.c/h contains code for directory access. See Chapter x, Directories for details about directory access.

fs_entry.c/h contains code for entry access. See Chapter x, Entries for details about entry access.

fs_file.c/h contains code for file access. See Chapter x, Files for details about file access.

fs_pool.c/h contains the code for pool management (used internally by µC/FS).

fs_sys.c/h contains the code for system driver management (used internally by µC/FS).

fs_unicode.c/h contains the code for handling Unicode strings (used internally by µC/FS).

3-9 µC/FS FAT FILESYSTEM SOURCE CODE

The files in these directories are available to µC/FS licensees (see Appendix H, Licensing Policy).

```
\Micrium
    \Software
        \uC-FS
            \FAT
                \fs_fat.c
                \fs_fat.h
                \fs_fat_dir.c
                \fs_fat_dir.h
                \fs_fat_entry.c
                \fs_fat_entry.h
                \fs_fat_fat12.c
                \fs_fat_fat12.h
                \fs_fat_fat16.c
                \fs_fat_fat16.h
                \fs_fat_fat32.c
                \fs_fat_fat32.h
                \fs_fat_file.c
                \fs_fat_file.h
                \fs_fat_journal.c
                \fs_fat_journal.h
                \fs_fat_lfn.c
                \fs_fat_lfn.h
                \fs_fat_sfn.c
                \fs_fat_sfn.h
                \fs_fat_type.h
```

\Micrium
This is where we place all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

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

\FAT
This directory contains the FAT system driver for µC/FS. All the files in this directory should be included in your build (assuming you have the source code).

3-10 µC/FS MEMORY DEVICE DRIVERS

These files are generic drivers to use with differenty memory devices.

```
\Micrium
    \Software
        \uC-FS
            \Dev
                \IDE
                    \fs_dev_ide.c
                    \fs_dev_ide.h
                    \BSP\Template
                        \fs_dev_ide_bsp.c
                \MSC
                    \fs_dev_msc.c
                    \fs_dev_msc.h
                \NAND
                    \fs_dev_nand.c
                    \fs_dev_nand.h
                    \PHY
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \Template
                            \fs_dev_nand_template.c
```

```
            \fs_dev_nand_template.h
    \BSP\Template
        \fs_dev_nand_bsp.c
    \BSP\Template (GPIO)
        \fs_dev_nand_bsp.c
    \BSP\Template (SPI GPIO)
        \fs_dev_nand_bsp.c
    \BSP\Template (SPI)
        \fs_dev_nand_bsp.c
\NOR
    \fs_dev_nor.c
    \fs_dev_nor.h
    \PHY
        \fs_dev_nor_amd_1x08.c
        \fs_dev_nor_amd_1x08.h
        \fs_dev_nor_amd_1x16.c
        \fs_dev_nor_amd_1x16.h
        \fs_dev_nor_intel.c
        \fs_dev_nor_intel.h
        \fs_dev_nor_sst25.c
        \fs_dev_nor_sst25.h
        \fs_dev_nor_sst39.c
        \fs_dev_nor_sst39.h
        \fs_dev_nor_stm25.c
        \fs_dev_nor_stm25.h
        \fs_dev_nor_stm29_1x08.c
        \fs_dev_nor_stm29_1x08.h
        \fs_dev_nor_stm29_1x16.c
        \fs_dev_nor_stm29_1x16.h
        \Template
            \fs_dev_nor_template.c
            \fs_dev_nor_template.h
    \BSP\Template
        \fs_dev_nor_bsp.c
    \BSP\Template (SPI GPIO)
        \fs_dev_nor_bsp.c
    \BSP\Template (SPI)
        \fs_dev_nor_bsp.c
```

```
\RAMDisk
    \fs_dev_ram.c
    \fs_dev_ram.h
\SD
    \fs_dev_sd.c
    \fs_dev_sd.h
    \Card
        \fs_dev_sd_card.c
        \fs_dev_sd_card.h
        \BSP\Template
            \fs_dev_sd_card_bsp.c
    \SPI
        \fs_dev_sd_spi.c
        \fs_dev_sd_spi.h
        \BSP\Template
            \fs_dev_sd_spi.bsp.c
\Template
    \fs_dev_template.c
    \fs_dev_template.h
```

\Micrium
This directory contains all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

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

\Dev
This is where you will find the device driver files for the storage devices you are planning on using.

\IDE
This directory contains the IDE/CF driver files.

fs_dev_ide.* are device driver for IDE devices. These files require a set of BSP functions to be defined in a file named **fs_dev_ide_bsp.c** to work with a particular hardware setup.

For more details on this driver, please refer to Chapter 11, “IDE/CF Driver” on page 124.

\MSC
This directory contains the MSC (Mass Storage Class - USB drives) driver files.

fs_dev_msc.* are device driver for MSC devices. This driver is designed to work with µC/USB host stack.

For more details on this driver, please refer to Chapter 13, “MSC Driver” on page 134.

\NAND
This directory contains the NAND driver files.

fs_dev_nand.* are the device driver for NAND devices. These files require a set of physical-layer functions (defined in a file name **fs_dev_nand_<physical type>.***) as well as BSP functions (to be defined in a file named **fs_dev_nand_bsp.c**) to work with a particular hardware setup.

For more details on this driver, please refer to Chapter 14, “NAND Flash Driver” on page 137.

\NOR
This directory contains the NOR driver files.

fs_dev_nor.* are the device driver for NOR devices. These files require a set of physical-layer functions (defined in a file name **fs_dev_nor_<physical type>.***) as well as BSP functions (to be defined in a file named **fs_dev_nor_bsp.c**) to work with a particular hardware setup.

For more details on this driver, please refer to Chapter 15, “NOR Flash Driver” on page 151.

\RAMDisk
This directory contains the RAM disk driver files.

fs_dev_ramdisk.* constitue the RAM disk driver.

For more details on this driver, please refer to Chapter 16, “RAM Disk Driver” on page 170.

\SD
This directory contains the SD/MMC driver files.

fs_dev_sd.* are device driver for SD devices. Theses files require to be used with either the **fs_dev_sd_spi.*** (for SPI/one-wire mode) or **fs_dev_sd_card.*** (for Card/4-wires mode) files. These files require a set of BSP functions to be defined in a file named either **fs_dev_sd_spi_bsp.c** or **fs_dev_sd_card_bsp.c** to work with a particular hardware setup.

For more details on this driver, please refer to Chapter 17, “SD/MMC Drivers” on page 174.

3-11 µC/FS PLATFORM-SPECIFIC SOURCE CODE

These files are provided by the µC/FS device driver developer. See Chapter 17, Porting µC/FS. However, the µC/FS source code is delivered with port examples.

```
\Micrium
    \Software
        \uC-FS
            \Examples
                \BSP
                    \Dev
                        <memory type>
                            <manufacturer>
                                <board name>
                                    \fs_dev_<memory type>_bsp.c
```

\Micrium
This directory contains all software components and projects provided by Micriµm.

\Software
This sub-directory contains all the software components and projects.

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

\Examples
This is where you will find the device driver BSP example files.

\Dev\<memory type>
This is where you will find the examples BSP for one memory type. The ‘<’ and ‘>’ are not part of the actual name. The memory types supported by µC/FS are the following: IDE, NAND, NOR, SD\CARD, SD\SPI.

\<manufacturer>
The name of the manufacturer of the evaluation board. The ‘<’ and ‘>’ are not part of the actual name.

3-12 µC/FS OS ABSTRACTION LAYER

This directory contains the RTOS abstraction layer which allows the use of µC/FS with nearly any commercial of in-house RTOS, or without any RTOS at all. The abstraction layer for the selected RTOS is placed in a sub-directory under OS as follows:

```
\Micrium
    \Software
        \uC-FS
            \OS
                \<rtos_name>
                    \fs_os.c
                    \fs_os.h
```

\Micrium
This directory contains all software components and projects provided by Micriμm.

\Software
This sub-directory contains all the software components and projects.

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

\OS
This is the main OS directory.

\<rtos_name>
This is the directory that contains the files to perform RTOS abstraction. Note that files for the selected RTOS abstraction layer must always be named **fs_os.***.

µC/FS has been tested with µC/OS-II, µC/OS-III and without an RTOS. The RTOS layer files are found in the following directories:

```
\Micrium\Software\uC-Clk\OS\None\fs_os.*

\Micrium\Software\uC-Clk\OS\Template\fs_os.*

\Micrium\Software\uC-Clk\OS\uCOS-II\fs_os.*

\Micrium\Software\uC-Clk\OS\uCOS-III\fs_os.*
```

3-13 SUMMARY

Below is a summary of all the directories and files involved in a µC/FS-based project. The '<-Cfg' on the far right indicates that these files are typically copied into the application (i.e., project) directory and edited based on project requirements.

```
\Micrium
    \Software
        \EvalBoards
            \<manufacturer>
                \<board name>
                    \<compiler>
                        \<project name>
                            \app.c
                            \app.h
                            \other
                        \BSP
                            \bsp.c
                            \bsp.h
```

```
                    \other
        \CPU
            \<manufacturer>
                \<architecture>
                    \*.*
        \uC-FS
            \APP\Template
                \fs_app.c                              <-Cfg
                \fs_app.h                              <-Cfg
            \CFG\Template
                \fs_cfg.h                              <-Cfg
            \Dev
                \IDE
                    \fs_dev_ide.c
                    \fs_dev_ide.h
                    \BSP\Template
                        \fs_dev_ide_bsp.c              <-Cfg
                \MSC
                    \fs_dev_msc.c
                    \fs_dev_msc.h
                \NAND
                    \fs_dev_nand.c
                    \fs_dev_nand.h
                    \PHY
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \fs_dev_nand_0512_x08.c
                        \fs_dev_nand_0512_x08.h
                        \Template
                            \fs_dev_nand_template.c
                            \fs_dev_nand_template.h
                    \BSP\<template>
                        \fs_dev_nand_bsp.c             <-Cfg
                \NOR
```

```
    \fs_dev_nor.c
    \fs_dev_nor.h
    \PHY
        \fs_dev_nor_amd_1x08.c
        \fs_dev_nor_amd_1x08.h
        \fs_dev_nor_amd_1x16.c
        \fs_dev_nor_amd_1x16.h
        \fs_dev_nor_intel.c
        \fs_dev_nor_intel.h
        \fs_dev_nor_sst25.c
        \fs_dev_nor_sst25.h
        \fs_dev_nor_sst39.c
        \fs_dev_nor_sst39.h
        \fs_dev_nor_stm25.c
        \fs_dev_nor_stm25.h
        \fs_dev_nor_stm29_1x08.c
        \fs_dev_nor_stm29_1x08.h
        \fs_dev_nor_stm29_1x16.c
        \fs_dev_nor_stm29_1x16.h
        \Template
            \fs_dev_nor_template.c        <-Cfg
            \fs_dev_nor_template.h        <-Cfg
    \BSP\<template>
        \fs_dev_nor_bsp.c                 <-Cfg
\RAMDisk
    \fs_dev_ram.c
    \fs_dev_ram.h
\SD
    \fs_dev_sd.c
    \fs_dev_sd.h
    \Card
        \fs_dev_sd_card.c
        \fs_dev_sd_card.h
        \BSP\Template
            \fs_dev_sd_card_bsp.c         <-Cfg
    \SPI
        \fs_dev_sd_spi.c
        \fs_dev_sd_spi.h
```

```
          \BSP\Template
             \fs_dev_sd_spi.bsp.c          <-Cfg
    \Template
       \fs_dev_template.c                  <-Cfg
       \fs_dev_template.h                  <-Cfg
\FAT
    \fs_fat.c
    \fs_fat.h
    \fs_fat_dir.c
    \fs_fat_dir.h
    \fs_fat_entry.c
    \fs_fat_entry.h
    \fs_fat_fat12.c
    \fs_fat_fat12.h
    \fs_fat_fat16.c
    \fs_fat_fat16.h
    \fs_fat_fat32.c
    \fs_fat_fat32.h
    \fs_fat_file.c
    \fs_fat_file.h
    \fs_fat_journal.c
    \fs_fat_journal.h
    \fs_fat_lfn.c
    \fs_fat_lfn.h
    \fs_fat_sfn.c
    \fs_fat_sfn.h
    \fs_fat_type.h
\OS
    \<template>
       \fs_os.c                            <-Cfg
       \fs_os.h                            <-Cfg
    \<rtos_name>
       \fs_os.c
       \fs_os.h
\Source
    \fs_c
    \fs.h
    \fs_api.c
```

```
            \fs_api.h
            \fs_buf.c
            \fs_buf.h
            \fs_cache.c
            \fs_cache.h
            \fs_cfg_fs.h
            \fs_ctr.h
            \fs_def.h
            \fs_dev.c
            \fs_dev.h
            \fs_dir.c
            \fs_dir.h
            \fs_entry.c
            \fs_entry.h
            \fs_err.c
            \fs_err.h
            \fs_file.c
            \fs_file.h
            \fs_partition.c
            \fs_partition.h
            \fs_pool.c
            \fs_pool.h
            \fs_sys.c
            \fs_sys.h
            \fs_type.h
            \fs_unicode.c
            \fs_unicode.h
            \fs_util.c
            \fs_util.h
            \fs_vol.c
            \fs_vol.h
        \OS
            \<architecture>
                \<compiler>
                    \os_cpu.h
                    \os_cpu_a.asm
                    \os_cpu_c.c
    \uC-CPU
```

```
        \cpu_core.c
        \cpu_core.h
        \cpu_def.h
        \Cfg\Template
            \cpu_cfg.h                              <-Cfg
        \<architecture>
            \<compiler>
                \cpu.h
                \cpu_a.asm
                \cpu_c.c
    \uC-Clk
        \Cfg
            \Template
                \clk_cfg.h                          <-Cfg
        \OS
            \<rtos_name>
                \clk_os.c
        \Source
            \clk.c
            \clk.h
    \uC-CRC
        \Cfg
            \Template
                \crc_cfg.h                          <-Cfg
        \Ports
            \<architecture>
                \<compiler>
                    \ecc_bch_4bit_a.asm
                    \ecc_bch_8bit_a.asm
                    \ecc_hamming_a.asm
                    \edc_crc_a.asm
        \Source
            \edc_crc.h
            \edc_crc.c
            \ecc_hamming.h
            \ecc_hamming.c
            \ecc_bch_8bit.h
            \ecc_bch_8bit.c
```

```
        \ecc_bch_4bit.h
        \ecc_bch_4bit.c
        \ecc_bch.h
        \ecc_bch.c
        \ecc.h
\uC-LIB
    \lib_ascii.c
    \lib_ascii.h
    \lib_def.h
    \lib_math.c
    \lib_math.h
    \lib_mem.c
    \lib_mem.h
    \lib_str.c
    \lib_str.h
    \Cfg\Template
        \lib_cfg.h                              <-Cfg
```

Chapter

4

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.

4-1 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.

4-2 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);
```

Here, "**vol:**" is the volume name. µC/FS imposes no 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
```

4-3 µC/FS FILE AND DIRECTORY NAMES AND PATHS

Files and directories 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 Appendix E, "FS_CFG_WORKING_DIR_EN" on page 529). 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.

4-4 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:

```
FullNameLenmax = VolNameLenmax + PathNameLenmax
```

Figure 2-3 demonstrates these definitions.

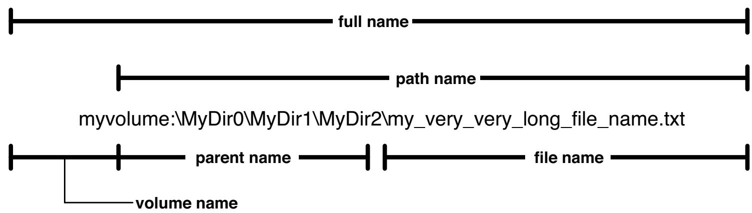


Figure 4-1 **File, path and volume name lengths**

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.

```
ParentNameLenmax = PathNameLenmax - FileNameLenmin - 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.

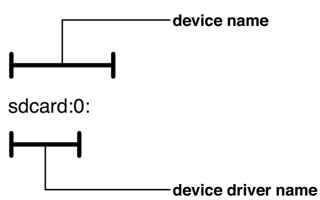


Figure 4-2 **Device and device driver name lengths**

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.

4-5 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 4-1 **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 Appendix E, “FS_CFG_MAX_PATH_NAME_LEN” on page 530).

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 4-2 **RAM Characteristics**

§ The number of tasks that use relative path names

See also section 10-1-1 “Driver Characterization” on page 121 for ROM/RAM characteristics of file system suite drivers.

Chapter 5

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 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 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 5-1) without requiring ambiguous terminology.

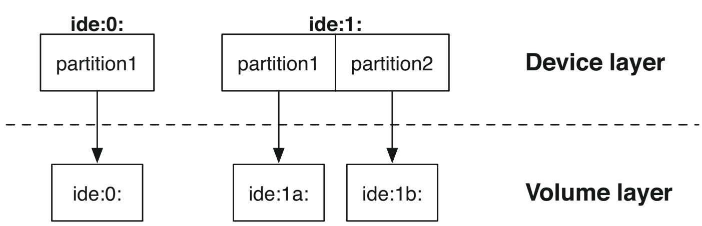


Figure 5-1 **Device and volume architecture.**

5-1 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 ivides the final unallocated portion of the device into two parts, so that a volume could be located on each (see section 5-4 "Partitions" on page 72).

- 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 4-2 "µC/FS Device and Volume Names" on page 61.

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.

Function	Description
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 5-1 **Device API functions**

5-2 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.

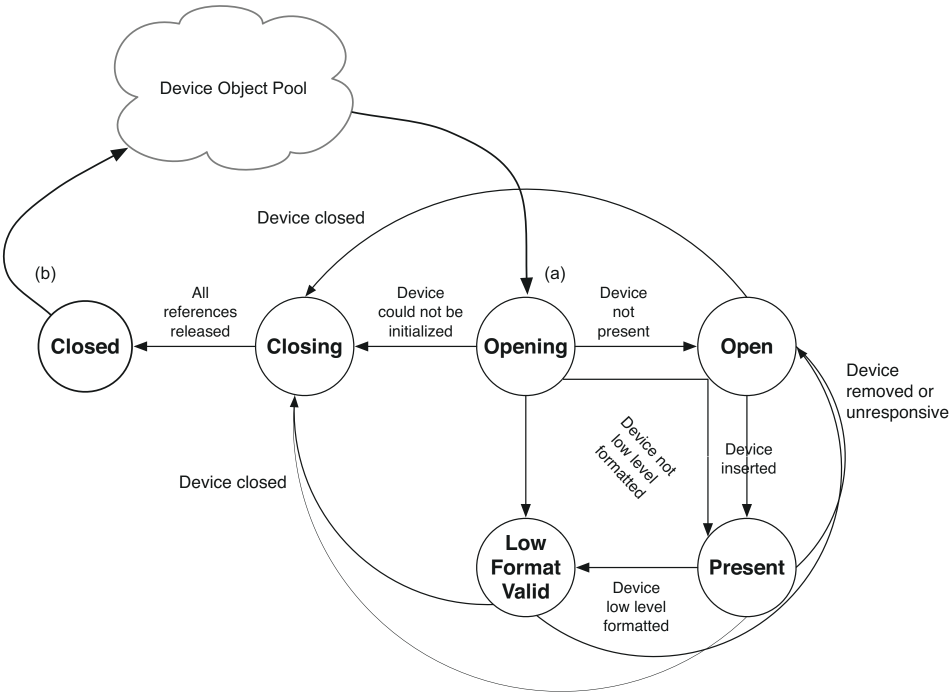


Figure 5-2 **Device state transition.**

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.

- 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.

5-3 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:

```
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).

5-4 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 5-4.

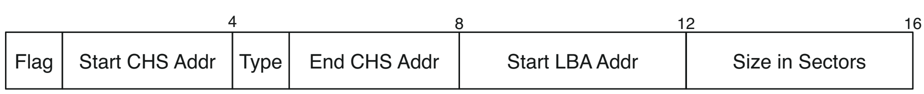


Figure 5-3 **Partition entry format.**

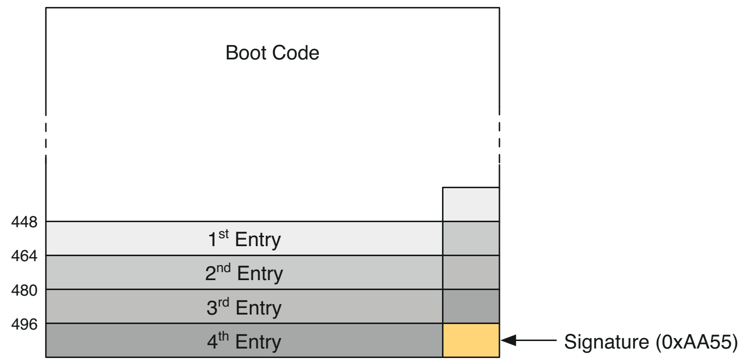


Figure 5-4 **Master Boot Record.**

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 5-5. 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.

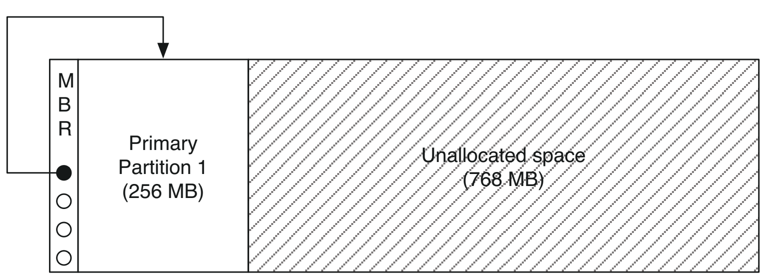


Figure 5-5 **Device after partition initialization.**

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 5-6.

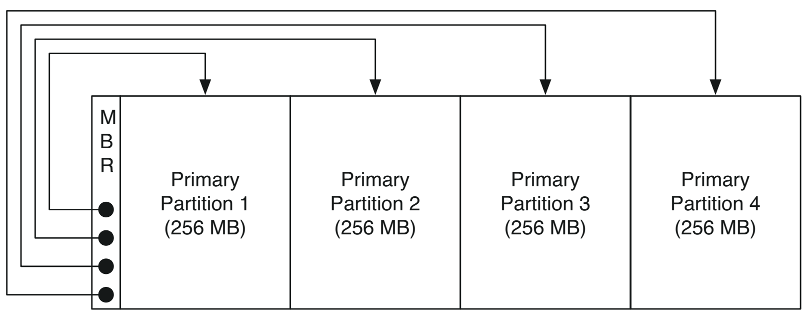


Figure 5-6 **Device after four partitions have been created.**

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.

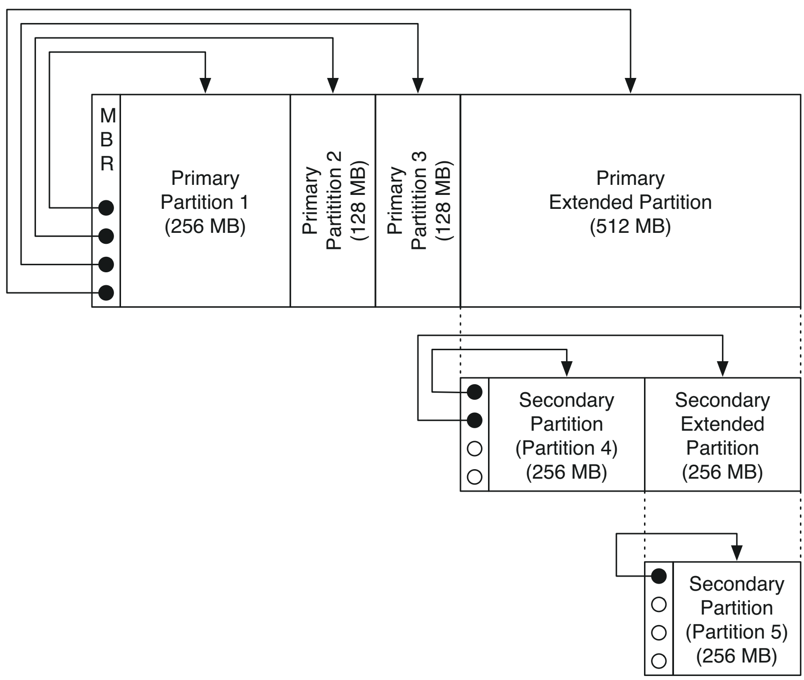


Figure 5-7 **Device with five partitions.**

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

5-5 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 4-2 "µC/FS Device and Volume Names" on page 61. For FAT-specific volume functions, see Chapter 9, "File Systems: FAT" on page 109.

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

Function	Description	Valid for Unmounted Volume?
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 5-2 **Volume API Functions**

5-6 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 5-2 “Using Devices” on page 69 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_SYS_CFG structure as the second argument. For more information about the FS_FAT_SYS_CFG structure, see section D-8 “FS_FAT_SYS_CFG” on page 520.

5-7 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 5-8) 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).

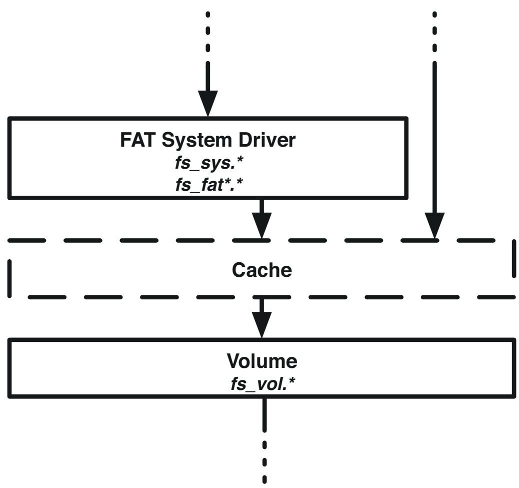


Figure 5-8 **Volume cache architecture.**

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.

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 5-3 **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.

5-7-1 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 vol cache API
                   (void             *)&CACHE_BUF[0],             <-- pointer to the cache buf
                   (CPU_INT32U        ) CACHE_BUF_LEN,            <-- cache buf 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.
```

Listing 5-1 **Cache**

L5-1(1) Percent of cache buffer dedicated to management sectors.

L5-1(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..

5-7-2 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 7-1-3 "Configuring a File Buffer" on page 101), 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

6

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.

6-1 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_FILEs) 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_DIRs). 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 4-3 “μC/FS File and Directory Names and Paths” on page 62.

Function	POSIX Equivalent		Function	POSIX Equivalent
fs_asctime_r()	asctime_r()		fs_ftruncate()	ftruncate()
fs_chdir()	chdir()		fs_ftrylockfile()	ftrylockfile()
fs_clearerr()	clearerr()		fs_funlockfile()	funlockfile()
fs_closedir()	closedir()		fs_fwrite()	fwrite()
fs_ctime_r()	ctime_r()		fs_getcwd()	getcwd()
fs_fclose()	fclose()		fs_localtime_r()	localtime_r()
fs_feof()	feof()		fs_mkdir()	mkdir()
fs_ferror()	ferror()		fs_mktime()	mktime()
fs_fflush()	fflush()		fs_rewind()	rewind()
fs_fgetpos()	fgetpos()		fs_opendir()	opendir()
fs_flockfile()	flockfile()		fs_readdir_r()	readdir_r()
fs_fopen()	fopen()		fs_remove()	remove()
fs_fread()	fread()		fs_rename()	rename()

Function	POSIX Equivalent		Function	POSIX Equivalent
fs_fseek()	fseek()		fs_rmdir()	rmdir()
fs_fsetpos()	fsetpos()		fs_setbuf()	setbuf()
fs_fstat()	fstat()		fs_setvbuf()	setvbuf()
fs_ftell()	ftell()		fs_stat()	stat()

Table 6-1 **POSIX API functions.**

6-2 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 root directory of the default volume is used.

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 F, "Shell Commands" on page 535).

6-3 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.

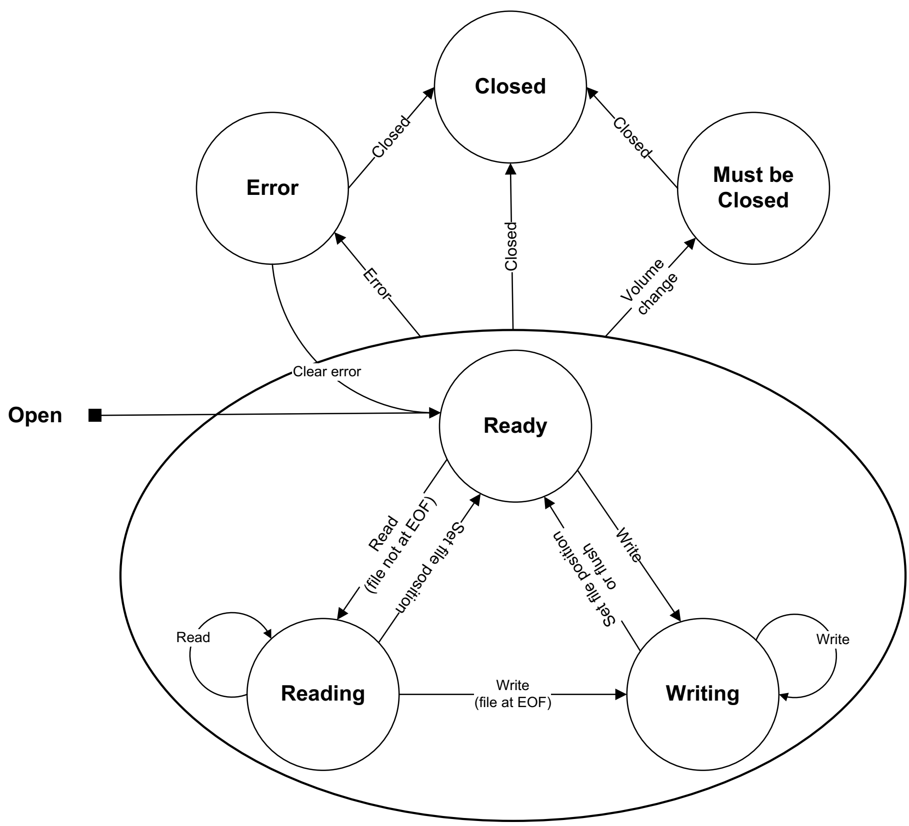


Figure 6-1 **File state transitions.**

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.

6-3-1 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 4-3 "µC/FS File and Directory Names and Paths" on page 62). 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 6-2 **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 6-1.

```
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"));
    }
    .
    .
    .
}
```

Listing 6-1 **Example file read.**

L6-1(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()`**.

L6-1(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.

6-3-2 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 6-2.

```
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.          */
    .
    .
    .
}
```

Listing 6-2 **Example file position set/get.**

6-3-3 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.

```
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.           */
    }
    .
    .
    .
}
```

Listing 6-3 **Example file buffer usage.**

L6-3(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.

L6-3(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.

6-3-4 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.

6-3-5 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()`.

```
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.               */
    }
    .
    .
    .
}
```

Listing 6-4 **Example file lock usage.**

L6-4(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.

6-4 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.

Figure 6-2 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.

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

    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));
}
.
.
.
}
```

Listing 6-5 **Directory Listing Output (example)**

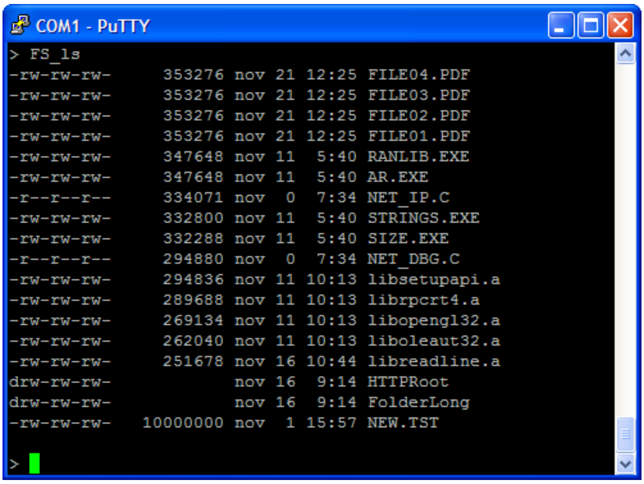


Figure 6-2 **Example directory listing.**

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 D-6 “FS_DIR_ENTRY (struct fs_dirent)” on page 517.

6-5 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 7 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 7-1.

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 7-3.

The entry functions and the **FSFile_Open()** function accept full file paths. For information about using file and path names, see section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

The functions listed in Table 7-1 and Table 7-3 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.

7-1 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_IsOpen()`	Determine whether a file is open or not.
`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 7-1 **File Access Functions**

7-1-1 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 7-2 **fopen() mode strings and mode equivalents**

7-1-2 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_ENTRY_INFO` structure has the following members:

- `Attrib` contains the file attributes (see section 7-2-1 "File and Directory Attributes" on page 104).
- `Size` is the size of the file, in octets.
- `DateTimeCreate` is the creation timestamp of the file.
- `DateAccess` is the access timestamp (date only) 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`, `DateAccess` and `DateTimeWr` are structures of type `CLK_TS_SEC`.

7-1-3 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 6-3-3 "Configuring a File Buffer" on page 91.

7-1-4 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 6-3-4 "Diagnosing a File Error" on page 93.

7-1-5 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 6-3-5 "Atomic File Operations Using File Lock" on page 93.

7-2 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 7-3 **Entry API Functions**

7-2-1 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:

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

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

`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.

7-2-2 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
                   FS_ENTRY_TYPE_FILE,    <-- means entry will be a file
                   DEF_NO,                <-- DEF_NO means creation NOT exclusive
                  &err);                  <-- return error
```

If the second argument, `entry_type`, is `FS_ENTRY_TYPE_DIR` 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.

7-2-3 DELETING FILES AND DIRECTORIES

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

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

The second argument, entry_type, restricts deletion to specific types. If it is **FS_ENTRY_TYPE_DIR**, 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 **FS_ENTRY_TYPE_FILE**, then the entry *must* be a file. If it is **FS_ENTRY_TYPE_ANY**, then the entry will be deleted whether it is a file or a directory.

Chapter

8

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 7-2 "Entry Access Functions" on page 103.

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 4-3 "µC/FS File and Directory Names and Paths" on page 62.

The functions listed in Table 8-1 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 Chapter 6, "POSIX API" on page 83. 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.

8-1 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.
FSDir_IsOpen()	Determine whether a directory is open or not.

Table 8-1 **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

```
FS_DIR *FSDir_Open (CPU_CHAR     *p_name_full,
                    FS_ERR       *p_err);

void    FSDir_Close(FS_DIR       *p_dir,
                    FS_ERR       *p_err);

void    FSDir_Rd   (FS_DIR       *p_dir,
                    FS_DIR_ENTRY *p_dir_entry,
                    FS_ERR       *p_err);
```

POSIX API Equivalent

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

int  fs_closedir
             (FS_DIR            *dirp);

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 6-4 “Directory Access Functions” on page 94.

Chapter

9

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.

9-1 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.

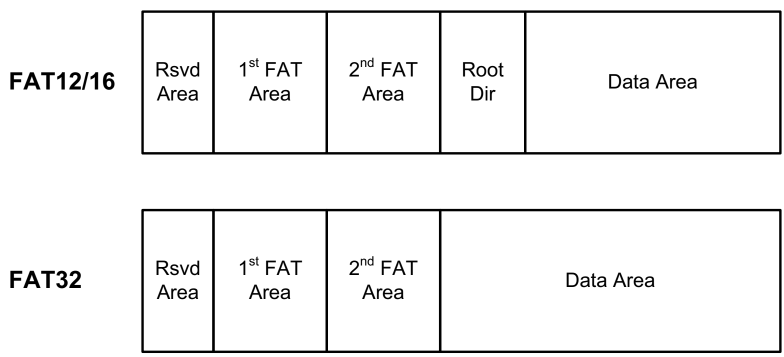


Figure 9-1 **FAT Volume Layout**

9-1-1 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 Appendix E, "Fat Configuration" on page 532, 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.

9-1-2 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 Appendix E, “Fat Configuration” on page 532, 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 Appendix E, “Feature Inclusion Configuration” on page 527).

9-1-3 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 9-2, 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.

<table>
<tr><td colspan="4" align="right">4</td><td colspan="4" align="right">8</td><td colspan="4" align="right">12</td><td colspan="4" align="right">16</td></tr>
<tr><td colspan="11">Name</td><td>Attr</td><td>NT res</td><td>Crt ms</td><td colspan="2">Creation Time</td></tr>
<tr><td colspan="2">Creation Date</td><td colspan="2">Access Date</td><td colspan="2">1st Cluster High</td><td colspan="2">Write Time</td><td colspan="2">Write Date</td><td colspan="2">1st Cluster Low</td><td colspan="4">File Size</td></tr>
</table>

Figure 9-2 **FAT Directory Entry (SFN Entry)**

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

0x42	'.'	'o'	'p'	0x0000	0xFFFF	0x0F	0x00	Chk sum	0xFFFF
0xFFFF	0xFFFF	0xFFFF	0xFFFF	0xFFFF	0x0000	0xFFFF	0xFFFF		

0x01	'a'	'b'	'c'	'd'	'e'	0x0F	0x00	Chk sum	'f'
'g'	'h'	'i'	'j'	'k'	0x0000	'l'	'm'		

'a'	'b'	'c'	'd'	'e'	'f'	'~'	'1'	'o'	'p'		0x00	0x00	Crt ms	Creation Time
Creation Date	Access Date	1st Cluster High	Write Time	Write Date	1st Cluster Low	File Size								

Figure 9-3 **LFN Directory Entry**

To extend FAT for longer names, Microsoft devised the LFN directory entry, as shown in Figure 9-2. 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 9-3 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).

4 8 12 16

Ord	Char 1	Char 2	Char 3	Char 4	Char 5	0x0F	0x00	Chk sum	Char 6
Char 7	Char 8	Char 9	Char 10	Char 11	0x0000	Char 12	Char 12		

4 8 12 16

0x42	'.'	'o'	'p'	0x0000	0xFFFF	0x0F	0x00	Chk sum	0xFFFF
0xFFFF	0xFFFF	0xFFFF	0xFFFF	0xFFFF	0x0000	0xFFFF	0xFFFF		

0x01	'a'	'b'	'c'	'd'	'e'	0x0F	0x00	Chk sum	'f'
'g'	'h'	'i'	'j'	'k'	0x0000	'l'	'm'		

'a'	'b'	'c'	'd'	'e'	'f'	'~'	'1'	'o'	'p'		0x00	0x00	Crt ms	Creation Time
Creation Date	Access Date	1st Cluster High	Write Time	Write Date	1st Cluster Low	File Size								

Figure 9-4 **SNF entry and LFN entries for file named "abcdefghijklm.op"**

9-1-4 FAT SYSTEM DRIVER ARCHITECTURE

As shown in Figure 9-2, 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 9-4. 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.

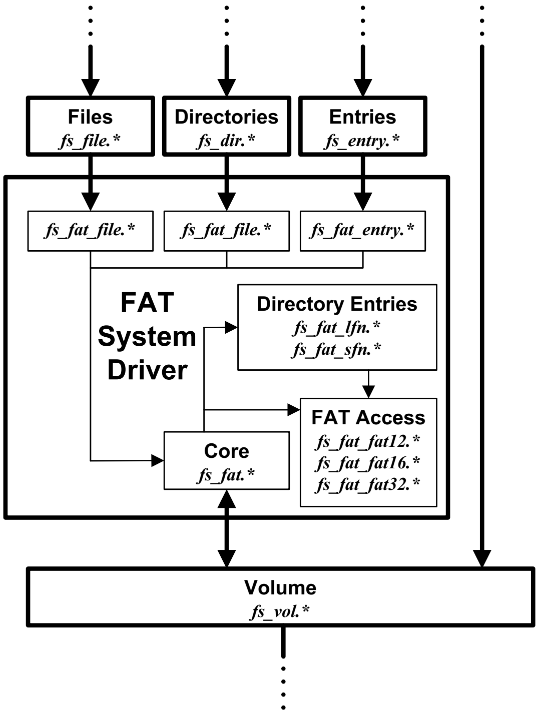


Figure 9-5 **FAT system driver architecture**

9-2 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.

9-2-1 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 9-1. If the configuration is invalid, an error will be returned from `FSVol_Fmt()`. For more information about the `FS_FAT_SYS_CFG` structure, see Appendix D, “FS_FAT_SYS_CFG” on page 520.

```
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"));
    }
    .
    .
    .
}
```

Listing 9-1 **Example device format**

9-2-2 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.

9-2-3 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.

```
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 9-2 **Opening the journal**

```
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.      */
    .
}
```

Listing 9-3 **Starting and stopping journaling**

Chapter

10

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').

10-1 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.
- NAND driver. The NAND flash driver support parallel (typically ONFI-compliant) and serial (typically Atmel Dataflash) NAND flash devices.
- 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 10-1 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	Reference
IDE/CF	`"ide:"`	`FSDev_IDE`	Chapter 11, on page 124
MSC	`"msc:"`	`FSDev_MSC`	Chapter 13, on page 134
NAND	`"nand:"`	`FSDev_NAND`	Chapter 14, on page 137
NOR	`"nor:"`	`FSDev_NOR`	Chapter 15, on page 151
RAM disk	`"ram:"`	`FSDev_RAM`	Chapter 16, on page 170

Driver	Driver Name	Driver API Structure Name	Reference
SD/MMC	`"sd:" / "sdcard:"`	`FSDev_SD_SPI / FSDev_SD_Card`	Chapter 17, on page 174

Table 10-1 **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. Appendix C, "Device Driver" on page 394 describes how to do this.

10-1-1 DRIVER CHARACTERIZATION

Typical ROM requirements are summarized in Table 10-2. The ROM data were collected on IAR EWARM v5.50 with high size optimization.

Driver	ROM, Thumb Mode	ROM, ARM Mode
IDE/CF *	3.6 kB	5.2 kB
MSC**	1.2 kB	1.6 kB
NAND***	8.7 kB	12,1 kB
NOR***	10.9 kB	15.2 kB
RAM disk	0.9 kB	1.2 kB
SD/MMC CardMode*	5.9 kB	8.6 kB
SD/MMC SPI*	5.5 kB	7.9 kB

Table 10-2 **Driver ROM requirements**

* Not including BSP
**Not including µC/USB
***Not including physical-level driver or BSP

Typical RAM requirements are summarized in Table 10-3.

Driver	RAM (Overhead)	RAM (Per Device)
IDE/CF	8 bytes	24 bytes
MSC*	12 bytes	32 bytes
NAND	8 bytes	--- bytes
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 10-3 **Driver RAM requirements**

*Not including µC/USB
***See section 15-2 “Driver & Device Characteristics” on page 154.

Performance can vary significantly as a result of CPU and hardware differences, both as well as file system format. Table 10-4 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 10-4 **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

Chapter

11

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.

11-1 FILES AND DIRECTORIES

The files inside the IDE driver directory are outlined in this section; the generic file system files, outlined in Chapter 3, “Directories and Files” on page 28, 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 C-5 “IDE/CF Device BSP” on page 408 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`

11-2 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 11-1. 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 10-1-1 “Driver Characterization” on page 121.

```
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);
}
```

Listing 11-1 **Opening a IDE/CF device volume**

L11-1(1) Register the IDE/CF device driver.

L11-1(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 5-2 “Using Devices” on page 69 for more information).

L11-1(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.

L11-1(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 11-1 (if a sufficiently high trace level is configured). See section E-9 “Trace Configuration” on page 534 about configuring the trace level.

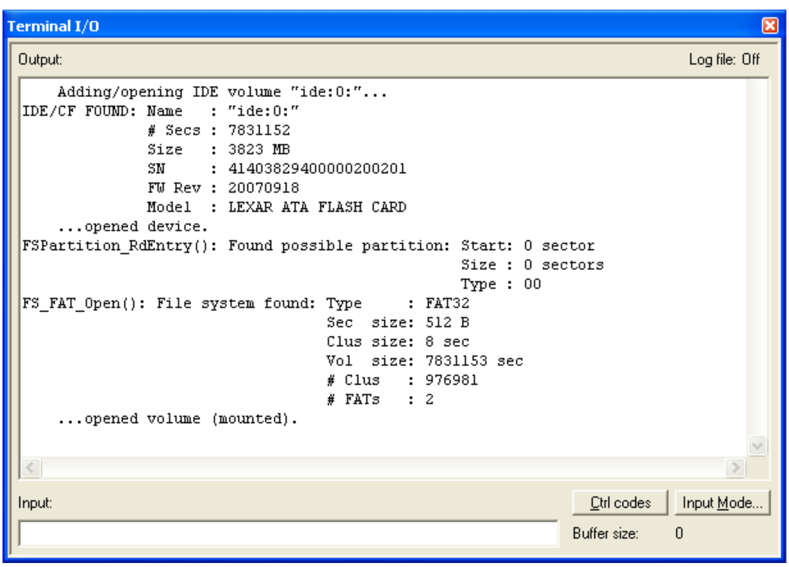


Figure 11-1 **IDE Detection Trace Output.**

11-2-1 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 11-2. 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.

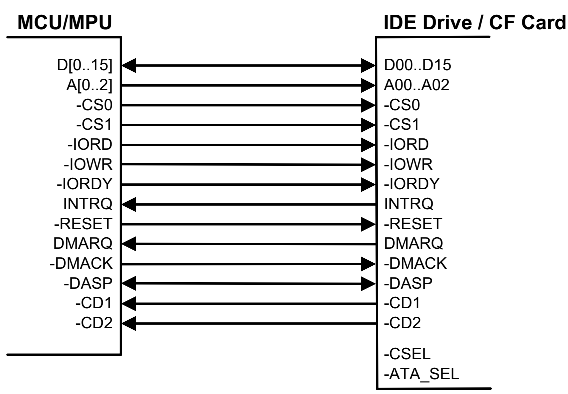


Figure 11-2 **True IDE (ATA) host/device connection**

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.

The host controls the device via 8 registers (see Figure 11-3). 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.

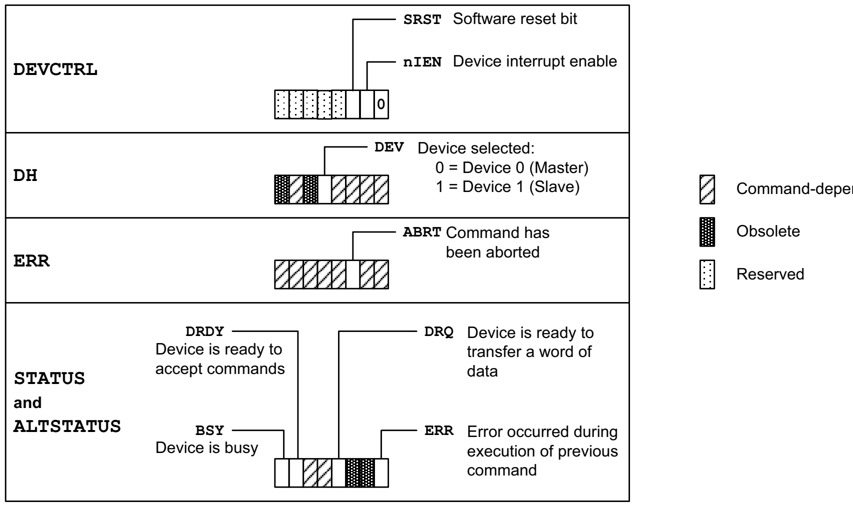


Figure 11-3 **Register definitions**

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

11-2-2 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. Pleaser refer to section C-5 "IDE/CF Device BSP" on page 408 for the details about implementing your own BSP.

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 11-1 **IDE BSP Functions**

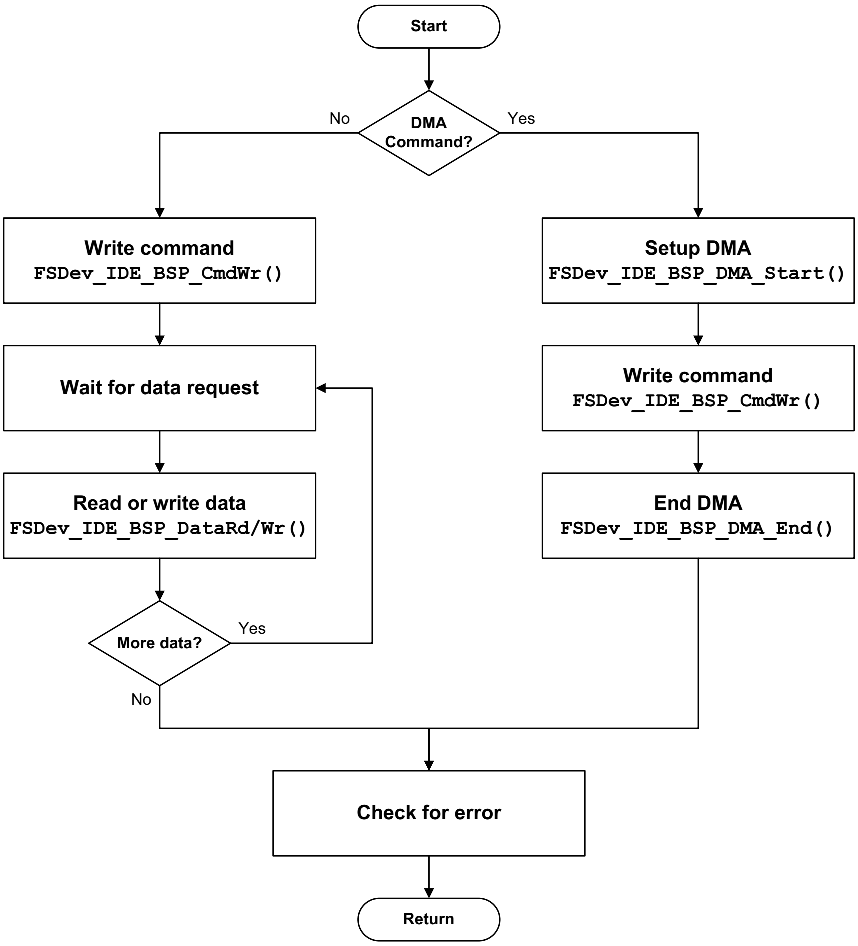


Figure 11-4 **Command execution**

Chapter

12

Logical Device Driver

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

Chapter

13

MSC Driver

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

13-1 FILES AND DIRECTORIES

The files inside the MSC driver directory are outlined in this section; the generic file-system files, outlined in Chapter 3, “Directories and Files” on page 28, 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.

13-2 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.
- fs_dev_msc.h.

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 13-1. 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 13-1.

ROM/RAM characteristics and performance benchmarks of the MSC driver can be found in section 10-1-1 “Driver Characterization” on page 121.

```
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 13-1 **Example µC/USB initialization**

```
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;
    }
}
```

Listing 13-2 **µC/USB MSC notification function**

If the file system and USB stack initialization succeed, the file system will produce the trace output as shown in Figure 13-1 (if a sufficiently high trace level is configured) when the a MSC device is connected. See section E-9 "Trace Configuration" on page 534 about configuring the trace level.

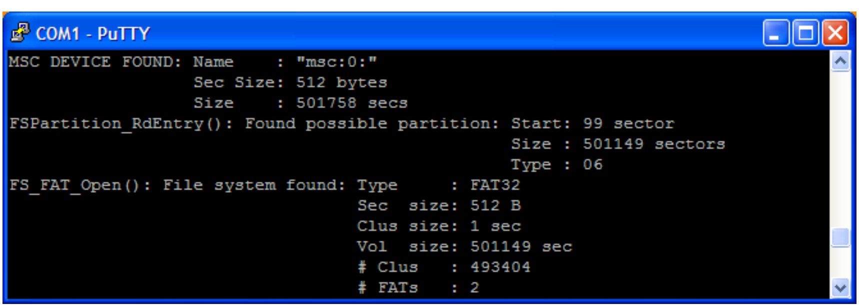


Figure 13-1 **MSC Detection Trace Output**

Chapter

14

NAND Flash Driver

NAND flash is a low-cost on-board storage solution. Typically, NAND flash have a multiplexed bus for address and data, resulting in a much lower pin count than parallel NOR devices. Their low price-per-bit and relatively high capacities often makes these preferable to NOR, though the higher absolute cost (because the lowest-capacity devices are at least 128-Mb) reverses the logic for applications requiring very little storage.

Standard storage media (like hard drives) or managed flash-based devices (like SD/MMC and CF cards) require relatively simple drivers that convert the file system's request to read or write a sector into a hardware transaction. The driver for a raw NAND flash (or raw NOR flash, for that matter) is more complicated. Flash is divided into large blocks (often 16-kB to 512-kB); however, the high-level software expects to read or write small sectors (512-bytes to 4096-bytes) atomically. The driver implements a small block abstraction (SBA) to conceal the device geometry from the file system. To aggravate matters, each block may be subjected to a finite number of erases only. A wear-leveling algorithm must be employed so that each block is used equally..

Device Category	**Capacity**	**Page Size**	**Block Size**	**Endurance**	**ECC**
Small-page SLC NAND Flash	128 Mb to 1 Gb	512 bytes	16 kB	100,000 erases/block	1-bit correction, 2-bit detection
Large-Page SLC NAND Flash	1 Gb to 4 Gb	2 kB or 4 kB	128 kB or 256 kB	100,000 erases/block	1-bit correction, 2-bit detection

Table 14-1 **NAND Flash Devices**

14-1 FILES AND DIRECTORIES

The files inside the NAND driver directory are outlined in this section; the generic file-system files, outlined in Chapter 3, "Directories and Files" on page 28, are also required.

\Micrium\Software\uC-FS\Dev
This directory contains device-specific files.

\Micrium\Software\uC-FS\Dev\NAND
This directory contains the NAND driver files.

fs_dev_nand.*
These files are device driver for NAND flash devices. This file requires a set of BSP functions be defined in a file named **fs_dev_nand_bsp.c** to work with a certain hardware setup.

.\BSP\Template\fs_dev_nand_bsp.c
This is a template BSP for traditional NAND devices accessed via a bus interface. See section C-7 "NAND Flash BSP" on page 440 for more information.

.\BSP\Template (GPIO)\fs_dev_nand_bsp.c
This is a template BSP for NAND devices accessed via GPIO. See section C-7 "NAND Flash BSP" on page 440 for more information.

.\BSP\Template (SPI GPIO)\fs_dev_nand_bsp.c
This is a template BSP for Atmel Dataflash devices accessed via GPIO (bit-banging). See section C-8 "NAND Flash SPI BSP" on page 450 for more information.

.\BSP\Template (SPI)\fs_dev_nand_bsp.c
This is a template BSP for Atmel Dataflash devices accessed via SPI. See section C-8 "NAND Flash SPI BSP" on page 450 for more information.

.\PHY
This directory contains physical-level drivers for specific NAND types:

`fs_dev_nand_0512x08.*`	512-byte page NAND, 8-bit data bus
`fs_dev_nand_2048x08.*`	2048-byte page NAND, 8-bit data bus
`fs_dev_nand_2048x16.*`	2048-byte page NAND, 16-bit data bus
`fs_dev_nand_at45.*`	Atmel AT45 serial flash

.\PHY\Template\fs_dev_nand_phy.c
This is a template for a physical-layer driver.

\Micrium\Software\uC-FS\Examples\BSP\Dev\NAND
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_nand_bsp.c`

14-2 DRIVER & DEVICE CHARACTERISTICS

All NAND devices 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; only an unprogrammed byte can have its bits changed from 1 to 0. Each block is divided into pages, which comprises a data area (often 512, 2048 or 4096 bytes) and a spare area (often 1/32 the size of the data area). The page is fundamentally the smallest programmable unit, but some devices allow several program operations per page between erases.

NAND flash experience occasional bit-errors, where one or more bits stored are flipped upon retrieval. An error-correcting code (ECC) is required so software can correct these bit-errors or take appropriate measures if too many errors occur. A single bit error-correcting code per 512 bytes of data is sufficient for single-level cell (SLC) flash.

The driver RAM requirement depends on flash parameters such as block size and run-time configurations such as sector size. Typical cases can be found in the datasheet.

14-3 USING A NAND DEVICE (SOFTWARE ECC)

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

- `fs_dev_nand.c`.
- `fs_dev_nand.h`.
- `fs_dev_nand_bsp.c` (located in the user application or BSP).
- A physical-layer driver, typically one provided in `\Micrium\Software\uC-FS\Dev\NAND\PHY`

The file `fs_dev_nand.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\NAND`
- `\Micrium\Software\uC-FS\Dev\NAND\PHY`

A single NAND volume is opened as shown in Listing 14-1. The file system initialization (`FS_Init()`) function must have previously been called and the NAND device driver, FSDev_NAND, registered (using FS_DevDrvAdd()).

ROM characteristics and performance benchmarks of the NAND driver can be found in section 10-1-1 “Driver Characterization” on page 121. The NAND driver also provides interface functions to perform low-level operations (see section A-9 “NAND Driver Functions” on page 340).

```
static  CPU_BOOLEAN  App_FS_AddNAND (void)
{
    FS_DEV_NAND_CFG  nand_cfg;
    FS_ERR           err;


    FS_DevDrvAdd((FS_DEV_API *)&FSDev_NAND,          /* (1)    */
                 (FS_ERR     *)&err);
    if ((err != FS_ERR_NONE) &&
        (err != FS_ERR_DEV_DRV_ALREADY_ADDED)) {
        APP_TRACE_DBG(("    ...could not add driver w/err = %d\r\n\r\n", &err));
        return (DEF_FAIL);
    }

                                                     /* (2)    */
    nand_cfg.BlkNbrFirst =  APP_CFG_FS_NAND_BLK_NBR_FIRST;
    nand_cfg.BlkCnt      =  APP_CFG_FS_NAND_BLK_CNT;

    nand_cfg.SecSize     =  APP_CFG_FS_NAND_SEC_SIZE;
    nand_cfg.RBCnt       =  APP_CFG_FS_NAND_RB_CNT;
    nand_cfg.PhyPtr      = (FS_DEV_NAND_PHY_API *)APP_CFG_FS_NAND_PHY_PTR;

    nand_cfg.BusWidth    =  APP_CFG_FS_NAND_BUS_WIDTH;
    nand_cfg.MaxClkFreq  =  APP_CFG_FS_NAND_MAX_CLK_FREQ;
```

```
                                                        /* (3)    */
    FSDev_Open(        "nand:0:",                       /*    (a) */
                (void *)&nand_cfg,                      /*    (b) */
                        &err);
    switch (err) {
        case FS_ERR_NONE:
             APP_TRACE_DBG(("    ...opened device.\r\n"));
             break;


        case FS_ERR_DEV_INVALID_LOW_FMT:
             APP_TRACE_DBG(("    ...opened device (not low-level formatted).\r\n"));
#if (FS_CFG_RD_ONLY_EN == DEF_DISABLED)
             FSDev_NAND_LowFmt("nand:0:", &err);        /* (4)    */
#endif
             if (err != FS_ERR_NONE) {
                APP_TRACE_DBG(("    ...low-level format failed.\r\n"));
                return (DEF_FAIL);
             }
             break;


        case FS_ERR_DEV:                                /* Device error.            */
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
        default:
             APP_TRACE_DBG(("    ...opening device failed w/err = %d.\r\n\r\n", err));
             return (DEF_FAIL);
    }
```

```
                                                    /* (5)    */
    FSVol_Open("nand:0:",                           /*    (a) */
               "nand:0:",                           /*    (b) */
                0,                                  /*    (c) */
                &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"));
#if (FS_CFG_RD_ONLY_EN == DEF_DISABLED)
             FSVol_Fmt("nand:0:", (void *)0, &err);  /* (6)    */
#endif
             if (err != FS_ERR_NONE) {
                APP_TRACE_DBG(("    ...format failed.\r\n"));
                return (DEF_FAIL);
             }
             break;


        case FS_ERR_DEV:                            /* Device error.            */
        case FS_ERR_DEV_IO:
        case FS_ERR_DEV_TIMEOUT:
        case FS_ERR_DEV_NOT_PRESENT:
             APP_TRACE_DBG(("    ...opened volume (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);
}
```

Listing 14-1 **Opening a NAND device volume**

L14-1(1) Register the NAND device driver `FSDev_`NAND.

L14-1(2) The NAND device configuration should be assigned. For more information about these parameters, see section D-3 “FS_DEV_NAND_CFG” on page 511.

L14-1(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 ("nand"), a single colon, an ASCII-formatted integer (the unit number) and another colon.

L14-1(4) **`FSDev_NAND_LowFmt()`** low-level formats a NAND. If the NAND 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.

`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.

`FSVol_Fmt()` formats a file system device. If the NAND has just been low-level formatted, there will be 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 NAND initialization succeeds, the file system will produce the trace output as shown in Figure 14-1 (if a sufficiently high trace level is configured). See section E-9 “Trace Configuration” on page 534 about configuring the trace level.

```
===================================================================
===================================================================
MEM module now initialized
FS Shell initialization succeeded
===================================================================
===================================================================

===================================================================
=                        FS INITIALIZATION                        =
===================================================================
Initializing FS...
    =========================================================
    Adding/opening NAND volume "nand:0:"...
NAND PHY AT45: Recognized device: Part nbr : Atmel AT45DB161D
                                  Page cnt : 8192
                                  Page size: 512
                                  Blk  cnt : 1024
                                  Blk  size: 4096
                                  Manuf ID : 0x1F
                                  Dev   ID : 0x27
NAND FLASH FOUND: Name       : "nand:0:"
                  Sec Size   : 512 bytes
                  Size       : 8192 secs
                  Replacemnt blks: 3
    ...opened device.
FSPartition_RdEntry(): Found possible partition: Start: 0 sector
                                                 Size : 0 sectors
                                                 Type : 00
FS_FAT_VolOpen(): File system found: Type      : FAT12
                                     Sec  size: 512 B
                                     Clus size: 8 sec
                                     Vol  size: 8192 sec
                                     # Clus    : 1018
                                     # FATs    : 2
    ...opened volume (mounted).
...init succeeded.
===================================================================
===================================================================
```

Figure 14-1 **NAND detection trace output**

14-3-1 DRIVER ARCHITECTURE

When used with a NAND device, the NAND driver is three layered, as depicted in the figure below. The generic NAND 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 the bus interface and access the NAND.

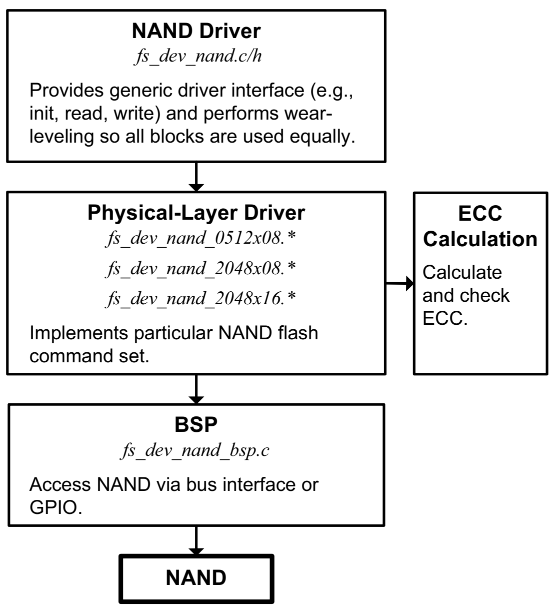
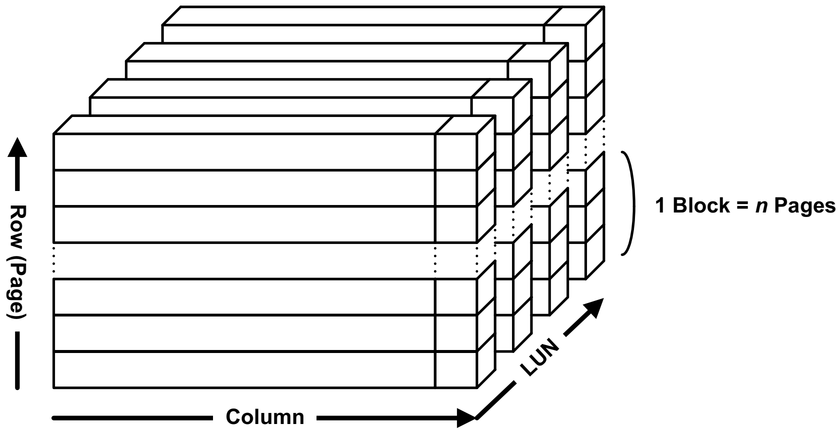


Figure 14-2 **NAND driver architecture**

14-3-2 HARDWARE

Parallel NAND devices typically connect to a host MCU/MPU via an external bus interface (EBI), with a 8 or 16 data lines, or via GPIO pins. Many silicon vendors offer NAND product lines; many new devices are conformant to the Open NAND Flash Interface (ONFI). A set of query information allows the µC/FS NAND driver physical-layer drivers to interface with these newer flash without configuration or modification; most older flash can be handled based purely on device ID.



Pin	Input/Output*	Description
`Chip Enable (nCE)`	O	Enables access to a specific chip. Several NANDs can be placed on the same bus if each has a separate chip enable.
`Command Latch Enable (CLE)`	O	Indicates that data is a command.
`Address Latch Enable (ALE)`	O	Indicates that data is an address.
`Read Enable (nRE)`	O	Enables serial data output from NAND.
`Write Enable (nWE)`	O	Controls latching of data input to NAND.
`Read/Busy (R/nB)`	I	Indicates status of NAND operation.
`Data Bus (D0...D7 or D0...D15)`	I/O	Used to write commands and addresses and to read/write data.

Table 14-2 **Pins, standard NAND**

*From perspective of CPU

14-3-3 NAND 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. Pleaser refer to section C-7 “NAND Flash BSP” on page 440 for the details about implementing your own BSP.

Function	Description
`FSDev_NAND_BSP_Open()`	Open (initialize) NAND bus interface.
`FSDev_NAND_BSP_Close()`	Close (uninitialize) NAND bus interface.
`FSDev_NAND_BSP_ChipSelEn()`	Enable NAND chip select.
`FSDev_NAND_BSP_ChipSecDis()`	Disable NAND chip select.
`FSDev_NAND_BSP_RdData()`	Read data from NAND.
`FSDev_NAND_BSP_WrAddr()`	Write address to NAND.
`FSDev_NAND_BSP_WrCmd()`	Write command to NAND.
`FSDev_NAND_BSP_WrData()`	Write data to NAND.
`FSDev_NAND_BSP_WaitWhileBusy()`	Wait while NAND is busy.

Table 14-3 **NAND BSP functions**

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

The remaining functions (`RdData(), WrAddr(), WrCmd(), WrData()`) read data from or write data to the NAND.

14-4 PHYSICAL-LAYER DRIVERS

The physical-layer drivers distributed with the NAND driver (see the table below) support a wide variety of flash devices from major vendors.

Driver API	Files	Description
`FSDev_NAND_0512x08`	*fs_dev_nand_0512x08.**	Supports 512-byte page SLC flash, 8-bit bus.
`FSDev_NAND_2048x08`	*fs_dev_nand_2048x08.**	Supports 2048-byte page SLC flash, 8-bit bus.
`FSDev_NAND_2048x16`	*fs_dev_nand_2048x16.**	Supports 2048-byte page SLC flash, 16-bit bus.
`FSDev_NAND_AT45`	*fs_dev_nand_at45.**	Supports various Atmel AT45 “DataFlash” serial devices.

Figure 14-3 **Physical-layer drivers**

14-4-1 FSDEV_NAND_0512X08

`FSDev_NAND_0512x08` supports small-page (512-byte) SLC NAND flash. The ECC is a 1-bit correct/2-bit detect code; this implementation uses a Hamming code. The sector size cannot exceed the page size, so the configured sector size MUST be 512-bytes.

14-4-2 FSDEV_NAND_2048X08, FSDEV_NAND_2048X16

`FSDev_NAND_2048x08` and FSDev_NAND_4096x08 support large-page (2048-byte) SLC NAND flash. The ECC is a 1-bit correct/2-bit detect code; this implementation uses a Hamming code. The sector size cannot exceed the page size, so the configured sector size MUST be less than 2048-bytes.

This physical-layer driver advertises its page size as the selected sector size, to take advantage of the partial page programming ability of SLC NAND. If a sector size of 512-bytes is used, the device MUST support at least four partial page programming operations between erases; if a sector size of 1024-bytes is used, the device MUST support at least two partial page programming operations between erases.

14-4-3 FSDEV_NAND_AT45

`FSDev_NAND_AT45` supports Atmel's AT45 serial flash memories ("DataFlash"), as described in various datasheets at Atmel (http://www.atmel.com). This driver has been tested with or should work with the devices in the table below.

While their underlying flash technology is NOR-type, the AT45-series devices are organized in a typical NAND-like way: each page of the device has a data area and a smaller spare area. No matter which AT45-series device is used, the physical-layer driver advertises its page size as 512-bytes; consequently, the driver MUST be configured with a 512-byte sector size.

Manufacturer	Device	Capacity	Device Page Size	Device Page Count
Atmel	AT45DB161D	16 Mb	512-byte	4096
Atmel	AT45DB321D	32 Mb	512-byte	8192
Atmel	AT45DB641D	64 Mb	1024-byte	8192

Table 14-4 **Supported AT45 serial flash**

Chapter

15

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. 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 15-1 briefly compares these two technologies; specific listings of supported devices are located in section 15-5 “Physical-Layer Drivers” on page 166.

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 15-1 **NOR Flash Devices**

15-1 FILES AND DIRECTORIES

The files inside the RAM disk driver directory are outlined in this section; the generic file-system files, outlined in Chapter 3, “Directories and Files” on page 28, 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.*
These files 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
This is a template BSP for traditional parallel NOR devices. See section C-10 “NOR Flash BSP” on page 459 for more information.

.\BSP\Template (SPI)\fs_dev_nor_bsp.c
This is a template BSP for serial (SPI) NOR devices. See section C-11 “NOR Flash SPI BSP” on page 466 for more information.

.\BSP\Template (SPI GPIO)\fs_dev_nor_bsp.c
This is a template BSP for serial (SPI) NOR devices using GPIO (bit-banging). See section C-11 “NOR Flash SPI BSP” on page 466 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
```

15-2 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.

15-3 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`.
- `fs_dev_nor.h`.
- `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 Table 15-1. 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 10-1-1 “Driver Characterization” on page 121. The NOR driver also provides interface functions to perform low-level operations (see section A-10 “NOR Driver Functions” on page 350).

```
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);
}
```

Listing 15-1 **Opening a NOR device volume**

L15-1(1) Register the NOR device driver `FSDev_NOR`.

L15-1(2) The NOR device configuration should be assigned. For more information about these parameters, see section D-4 “FS_DEV_NOR_CFG” on page 513.

L15-1(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.

L15-1(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.

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.

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 15-1 (if a sufficiently high trace level is configured). See section E-9 "Trace Configuration" on page 534 about configuring the trace level.

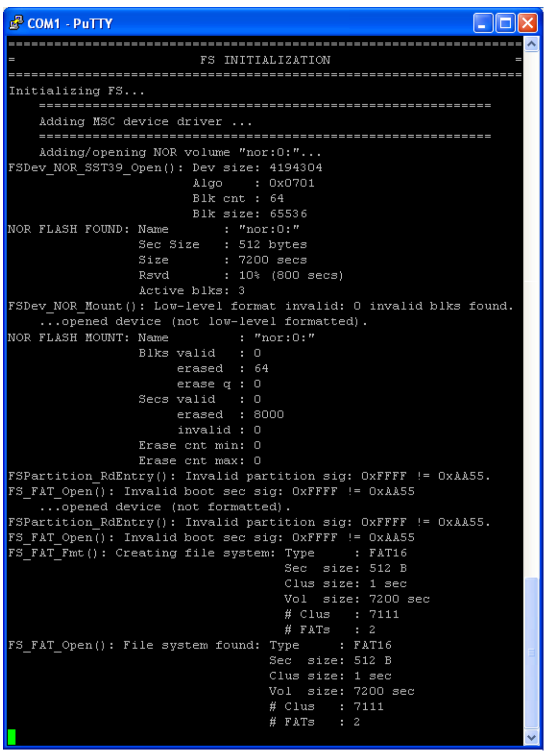


Figure 15-1 **NOR detection trace output**

15-3-1 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.

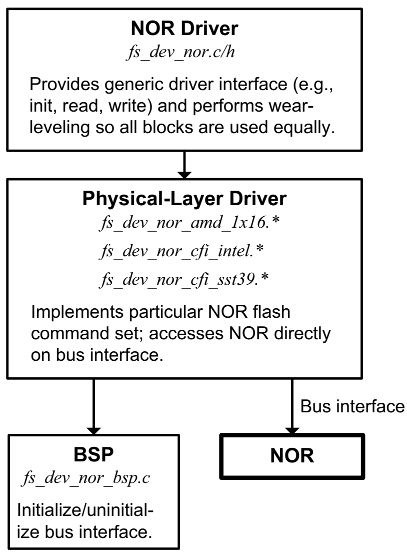


Figure 15-2 **NOR driver architecture (parallel NOR flash)**

15-3-2 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 = 2n
`0x2A`	2	Maximum number of bytes in multi-byte write = 2N
`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 15-2 **CFI query information**

Table 15-2 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 15-5-3

"FSDev_NOR_SST39" on page 168 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 15-3 **Common command sets**

15-3-3 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. Pleaser refer to section C-10 "NOR Flash BSP" on page 459 for the details about implementing your own BSP.

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 15-4 **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.

15-4 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.

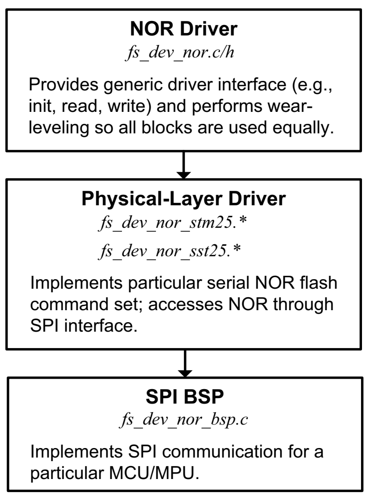


Figure 15-3 **NOR driver architecture (serial NOR flash)**

15-4-1 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 15-5, or similar, are common, and are often employed with the HOLD and WP pins held high (logic low, or inactive), as shown in Table 15-5. As with any SPI device, four signals are used to communicate with the host (CS, SI, SCK and SO).

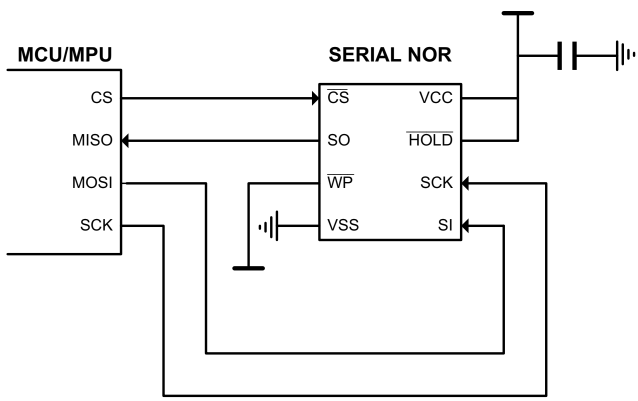


Figure 15-4 **Typical serial NOR connections**

15-4-2 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 C-11 on page 466.

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 15-5 **NOR SPI BSP Functions**

15-5 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 15-6 **Physical-layer drivers**

15-5-1 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.

15-5-2 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

15-5-3 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.

15-5-4 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 15-7 **Supported M25 serial flash**

15-5-5 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 15-8 **Supported SST25 serial flash**

Chapter

16

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.

16-1 FILES AND DIRECTORIES

The files inside the RAM disk driver directory are outlined in this section; the generic file-system files, outlined in Chapter 3, “Directories and Files” on page 28, 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.

16-2 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`.
- `fs_dev_ramdisk.h`.

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 . 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 10-1-1 “Driver Characterization” on page 121. For more information about the **`FS_DEV_RAM_CFG`** structure, see section D-5 “FS_DEV_RAM_CFG” on page 516.

```
#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);
}
```

Listing 16-1 **Opening a RAM disk volume**

L16-1(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.

L16-1(2) Register the RAM disk driver `FSDev_RAM`.

L16-1(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.

L16-1(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.

L16-1(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.

L16-1(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 16-1 (if a sufficiently high trace level is configured). See section E-9 “Trace Configuration” on page 534 about configuring the trace level.

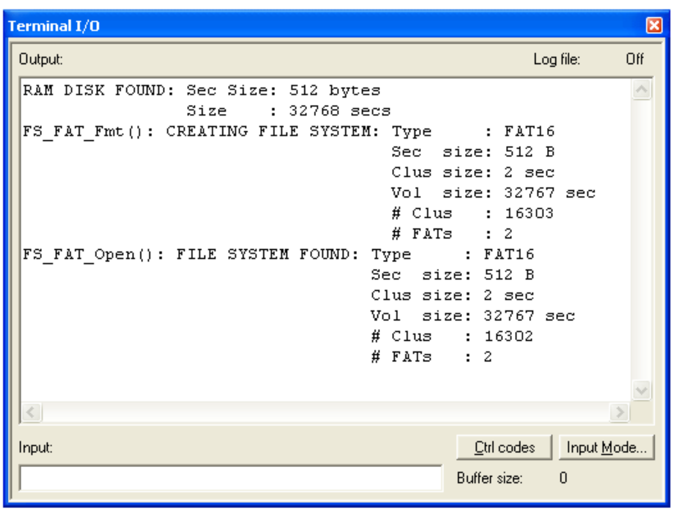


Figure 16-1 **RAM Disk Initialization Trace Output**

Chapter

17

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 17-1, 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	
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 17-1 **SD/MMC Devices**

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.

17-1 FILES AND DIRECTORIES

The files inside the SD/MMC driver directory is outlined in this section; the generic file-system files, outlined in Chapter 3, “Directories and Files” on page 28, 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 C-12 “SD/MMC Cardmode BSP” on page 467 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 C-13 “SD/MMC SPI mode BSP” on page 493 for more information.

`.\BSP\Template (GPIO)\fs_dev_sd_spi_bsp.c` is a template GPIO (bit-banging) BSP. See section C-13 “SD/MMC SPI mode BSP” on page 493 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`

17-2 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`.
- `fs_dev_sd.h`.
- `fs_dev_sd_card.c`.
- `fs_dev_sd_card.h`.
- `fs_dev_sd_card_bsp.c`.

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 17-1. 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 10-1-1 “Driver Characterization” on page 121. 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 “SD/MMC Driver Functions” on page 364).

```
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);
}
```

Listing 17-1 **Opening a SD/MMC device volume.**

L17-1(1) Register the SD/MMC CardMode device driver `FSDev_SD_Card`.

L17-1(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 5-2 "Using Devices" on page 69 for more information).

L17-1(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 17-1 (if a sufficiently high trace level is configured). See section E-9 “Trace Configuration” on page 534 about configuring the trace level.

```
COM1 - PuTTY
SD/MMC FOUND:  v1.x SD card
               Blk Size       : 512 bytes
               # Blks         : 1990656
               Max Clk        : 25000000 Hz
               Manufacturer ID: 0x27
               OEM/App ID     : 0x5048
               Prod Name      : SD01G
               Prod Rev       : 1.1
               Prod SN        : 0x701175A5
               Date           : 2/2007
FSPartition_RdEntry(): Found possible partition: Start: 249 sector
                                                 Size : 1990407 sectors
                                                 Type : 0B
FS_FAT_Open(): File system found: Type      : FAT32
                                  Sec  size: 512 B
                                  Clus size: 8 sec
                                  Vol  size: 1990407 sec
                                  # Clus    : 248310
                                  # FATs    : 2
```

Figure 17-1 **SD/MMC detection trace output**

17-2-1 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	Vss1	S	Supply voltage ground
4	VDD	S	Supply voltage

Pin	Name	Type	Description
5	CLK	I	Clock
6	VSS2	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 17-2 **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.

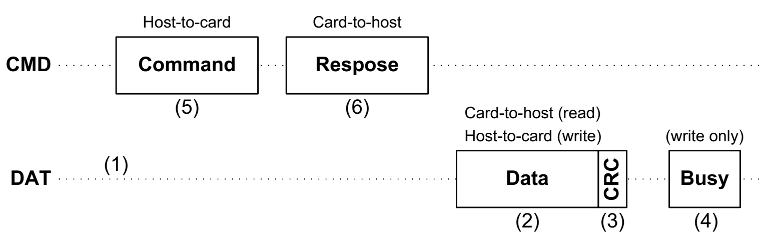


Figure 17-2 **SD/MMC communication sequence**

F17-2(1) When no data is being transmitted, data lines are held low.

F17-2(2) Data block is preceded by a start bit (‘0’); an end bit (‘1’) follows the CRC.

F17-2(3) The CRC is the 16-bit CCITT CRC.

F17-2(4) During the busy signaling following a write, DAT0 only is held low.

F17-2(5) See Figure 17-3 for description of the command format.

F17-2(6) See Figure 17-3 for description of the command format.

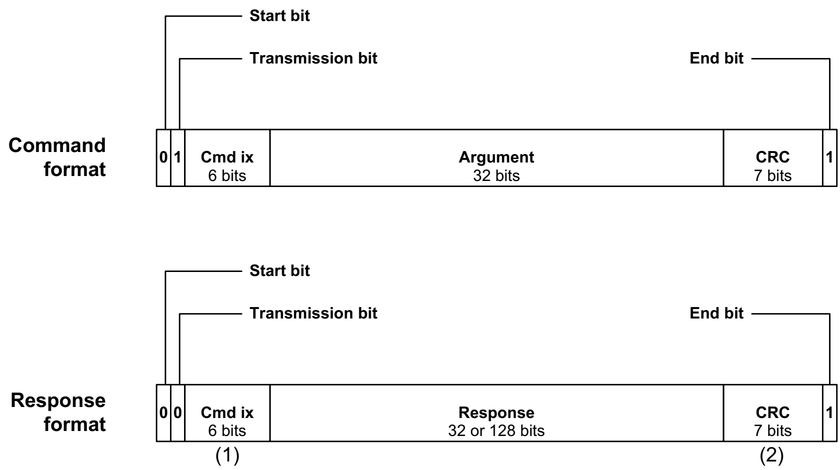


Figure 17-3 **SD/MMC command and response formats.**

F17-3(1) Command index is not valid for response formats R2 and R3.

F17-3(2) 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.

17-2-2 SD/MMC CARDMODE COMMUNICATION DEBUGGING

The SD/MMC cardmode driver accesses the hardware through a port (BSP). 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 Figure 17-5 and Figure 17-6. 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()`.

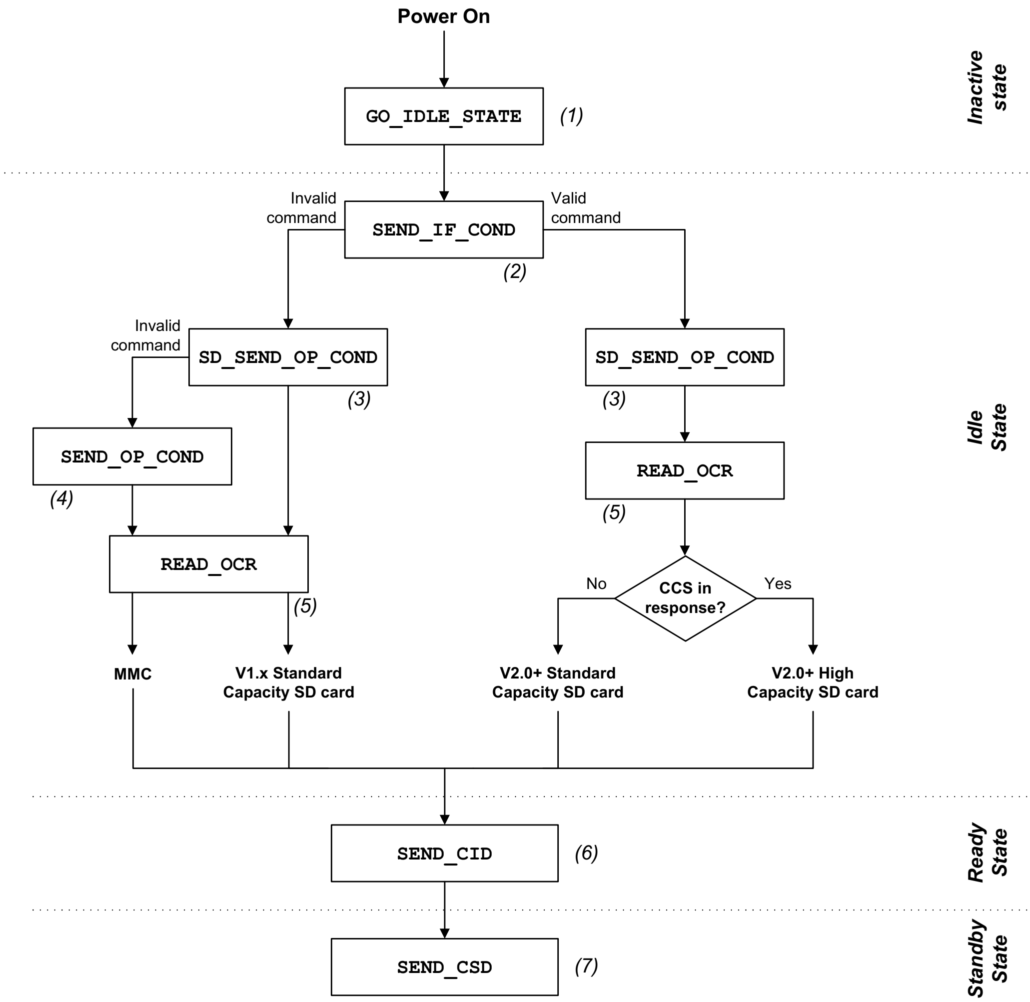


Figure 17-4 **Simplified SD/MMC cardmode initialization and state transitions**

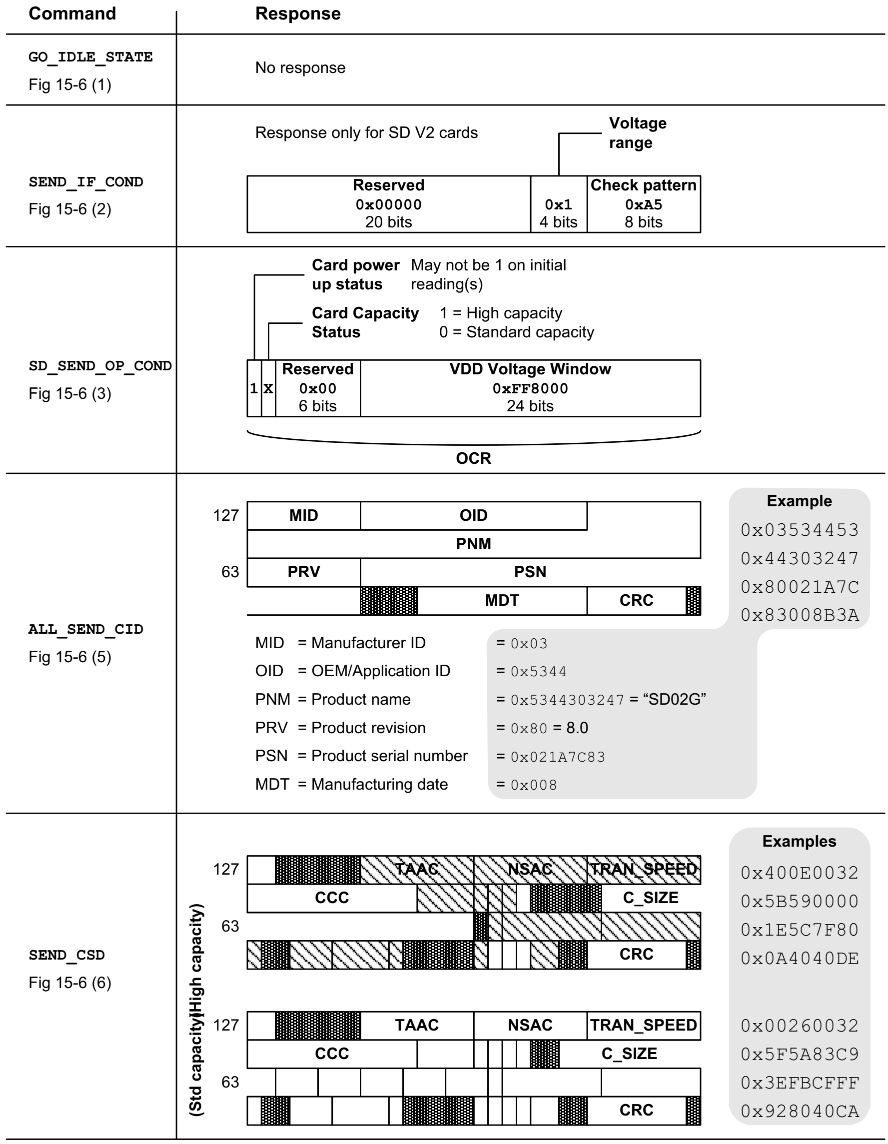


Figure 17-5 **Command responses (SD card)**

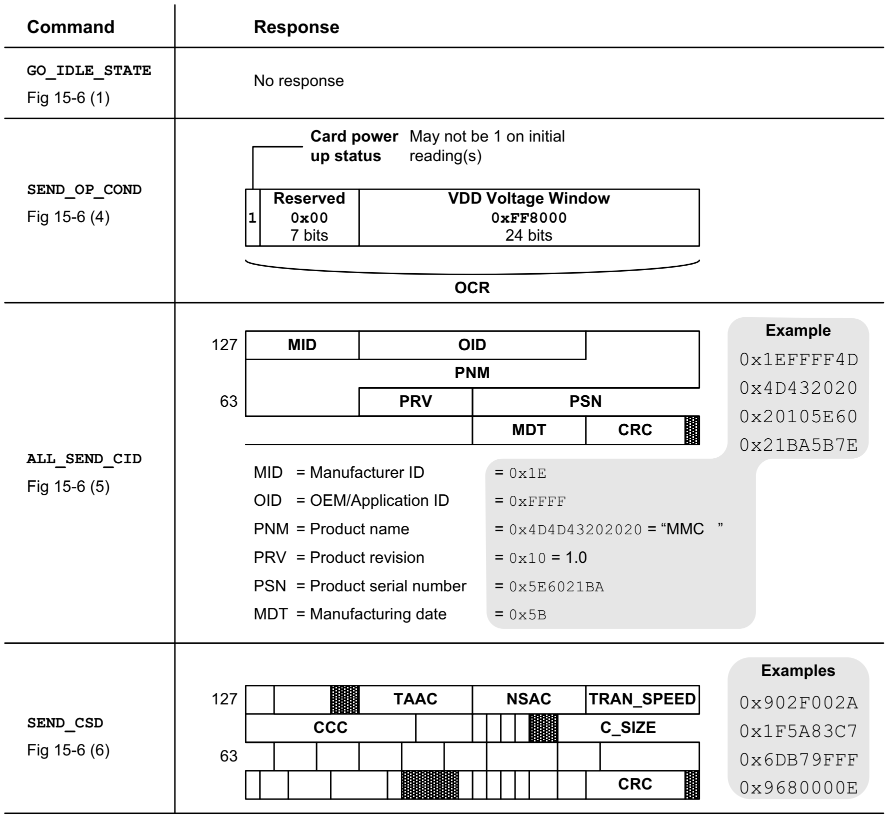


Figure 17-6 **Command responses (MMC card)**

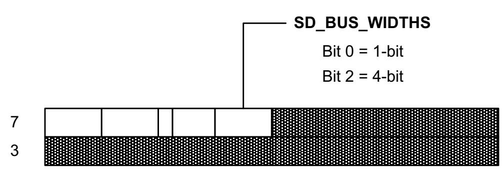


Figure 17-7 **SD SCR Register**

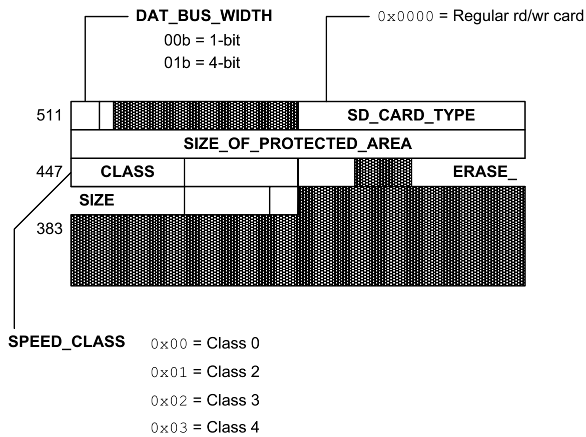


Figure 17-8 **SD Status**

17-2-3 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. Pleaser refer to section C-12 “SD/MMC Cardmode BSP” on page 467 for the details about implementing your own BSP.

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.

Function	Description
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 17-3 **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 17-9). 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.

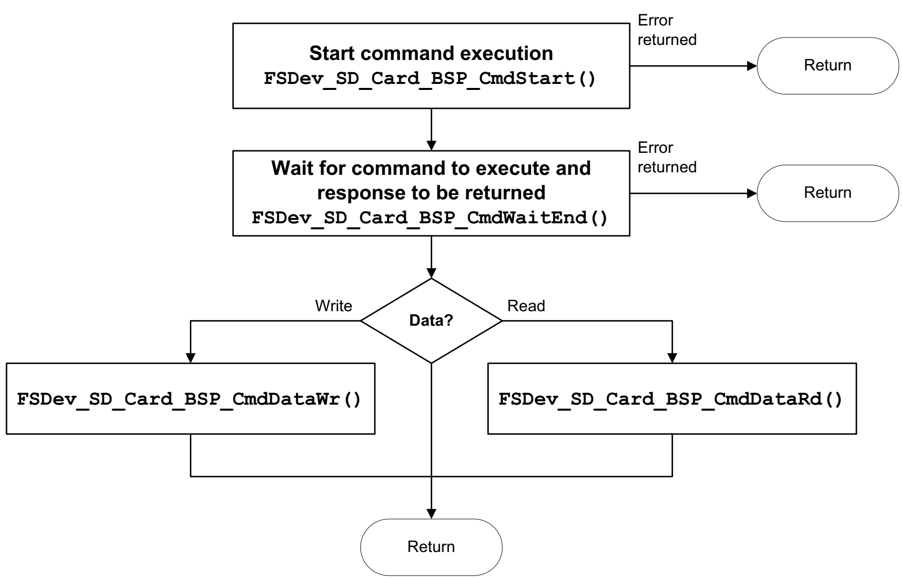


Figure 17-9 **Command execution**

17-3 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`.
- `fs_dev_sd.h`.
- `fs_dev_sd_spi.c`.
- `fs_dev_sd_spi.h`.
- `fs_dev_sd_spi_bsp.c`.

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 17-2. 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 10-1-1 “Driver Characterization” on page 121. 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 “SD/MMC Driver Functions” on page 364).

```
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);
}
```

Listing 17-2 **Opening a SD/MMC device volume**

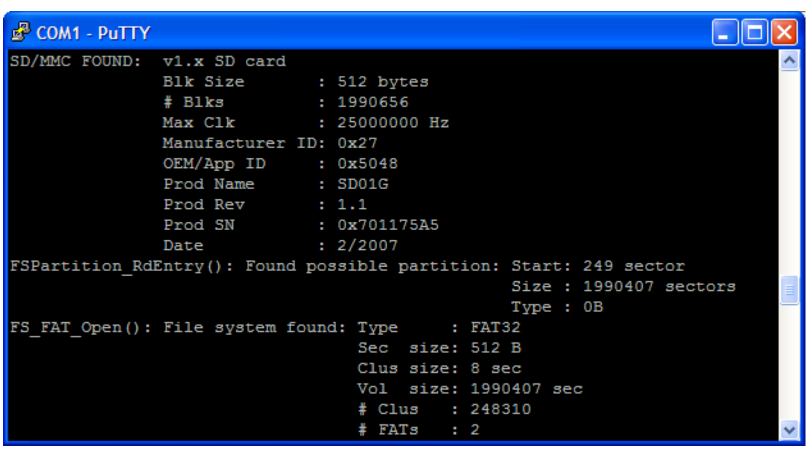
L17-2(1) Register the SD/MMC SPI device driver `FSDev_SD_SPI`.

L17-2(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 5-2 "Using Devices" on page 69 for more information).

L17-2(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 17-10 (if a sufficiently high trace level is configured). See section E-9 "Trace Configuration" on page 534 about configuring the trace level.




```
SD/MMC FOUND:  v1.x SD card
               Blk Size        : 512 bytes
               # Blks          : 1990656
               Max Clk         : 25000000 Hz
               Manufacturer ID: 0x27
               OEM/App ID      : 0x5048
               Prod Name       : SD01G
               Prod Rev        : 1.1
               Prod SN         : 0x701175A5
               Date            : 2/2007
FSPartition_RdEntry(): Found possible partition: Start: 249 sector
                                                 Size : 1990407 sectors
                                                 Type : 0B
FS_FAT_Open(): File system found: Type      : FAT32
                                  Sec  size: 512 B
                                  Clus size: 8 sec
                                  Vol  size: 1990407 sec
                                  # Clus    : 248310
                                  # FATs    : 2
```

Figure 17-10 **SD/MMC detection trace output**

17-3-1 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 17-4. 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	Vss1	S	Supply voltage ground
4	VDD	S	Supply voltage
5	CLK	I	Clock
6	VSS2	S	Supply voltage ground
7	DataOut	O	Card-to-host data and status

Table 17-4 **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.

17-3-2 SD/MMC SPI COMMUNICATION DEBUGGING

The SD/MMC SPI driver accesses the hardware through a port (SPI BSP) as described in section C-13 "SD/MMC SPI mode BSP" on page 493. 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. Initialization must succeed.

2 Read data. Data must be read from card, in both single- and multiple-block transactions.

3 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).

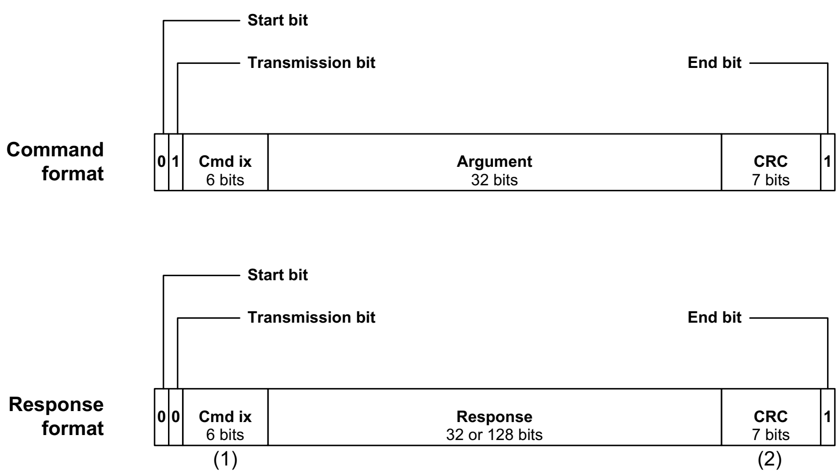


Figure 17-11 **SD/MMC SPI mode communication sequence**

F17-11(1) When no data is being transmitted, DataOut line is held high.

F17-11(2) During busy signaling, DataOut line is held low.

F17-11(3) 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.

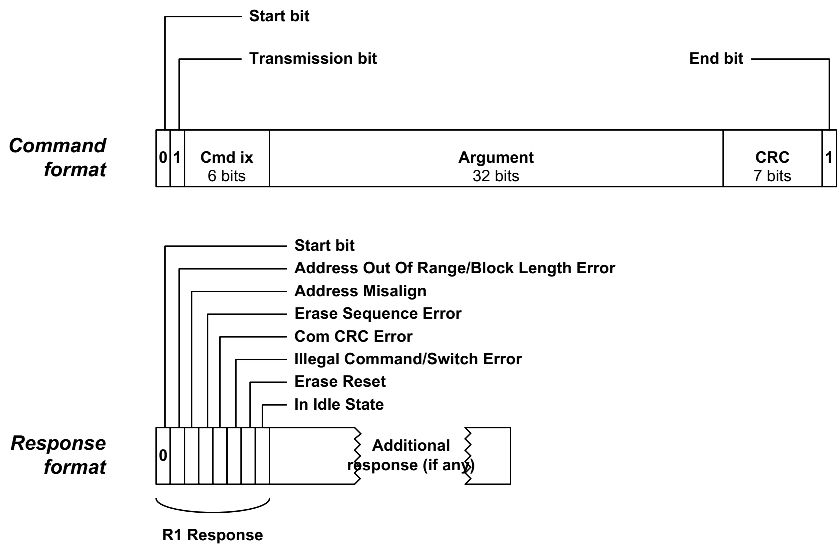


Figure 17-12 **SD/MMC SPI mode command and response formats**

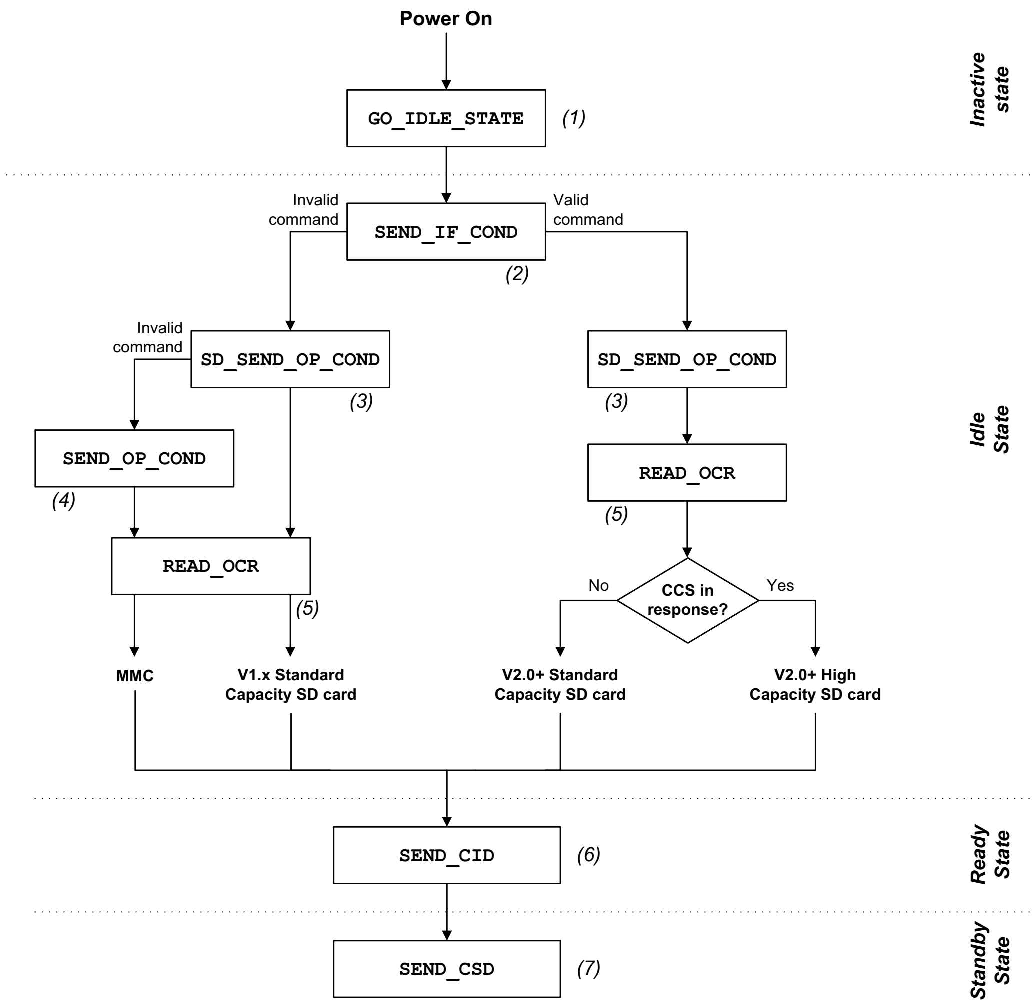


Figure 17-13 **Simplified SD/MMC SPI mode initialization and state transitions.**

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 Figure 17-5 and Figure 17-6), 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.

17-3-3 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 C-14 "SPI BSP" on page 494.

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 17-5 **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_F
Directory functions	FSDir_
Entry functions	FSEntry_
File functions	FSFile_
Time functions	FSTime_
Volume functions	FSVol_
NAND driver functions	FSDev_NAND_
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-1 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);
```

A-1-1 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:

- The name must be unique (e.g., a name that is not returned by any other device driver);
- The name must NOT include any of the characters: ':', '\' or '/'.
- The name must contain fewer than `FS_CFG_MAX_DEV_DRV_NAME_LEN` characters;
- 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).

A-1-2 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.

A-1-3 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.

A-1-4 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.

A-1-5 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-2 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);
```

A-2-1 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.

A-2-2 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) relative path that will be applied to the current working directory.

RETURNED VALUE

0, if no error occurs.

-1, otherwise

NOTES/WARNINGS

None.

A-2-3 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.

A-2-4 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.

A-2-5 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.

A-2-6 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.

A-2-7 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.

A-2-8 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.

A-2-9 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.

A-2-10 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.

A-2-11 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.

A-2-12 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 4-3 "µC/FS File and Directory Names and Paths" on page 62 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 section Table 7-2 "fopen() mode strings and mode equivalents" on page 100.

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.

A-2-13 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.

A-2-14 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.

A-2-15 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()`.

A-2-16 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.

A-2-17 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.

A-2-18 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()`.

A-2-19 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()`.

A-2-20 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.

A-2-21 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.

A-2-22 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.

A-2-23 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

```
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."));
    }

    .
    .
    .
}
```

A-2-24 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.

A-2-25 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 4-3 “µC/FS File and Directory Names and Paths” on page 62 for information about directory names.

RETURNED VALUE

Pointer to a directory, if NO errors.

Pointer to NULL, otherwise.

NOTES/WARNINGS

None.

A-2-26 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.

A-2-27 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 NOT 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

```
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."));
    }
    .
    .
    .
}
```

A-2-28 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 NOT 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

```
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.

A-2-29 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.

A-2-30 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

```
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.”));
    }

    .
    .
    .
}
```

A-2-31 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.

A-2-32 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-3 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 FS_SEC_QTY FS_ERR	*name_dev, partition_size, *p_err);
void FSDev_PartitionFind	(CPU_CHAR FS_PARTITION_NBR FS_PARTITION_ENTRY FS_ERR	*name_dev, partition_nbr, *p_partition_entry, *p_err);
void FSDev_PartitionInit	(CPU_CHAR FS_SEC_QTY FS_ERR	*name_dev, partition_size, *p_err);
void FSDev_Query	(CPU_CHAR FS_DEV_INFO FS_ERR	*name_dev, *p_info, *p_err);
void FSDev_Rd	(CPU_CHAR void FS_SEC_NBR FS_SEC_QTY FS_ERR	*name_dev, *p_dest, start, cnt, *p_err);
CPU_BOOLEAN FSDev_Refresh	(CPU_CHAR FS_ERR	*name_dev, *p_err);
void FSDev_Wr	(CPU_CHAR void FS_SEC_NBR FS_SEC_QTY FS_ERR	*name_dev, *p_src, start, cnt, *p_err);

A-3-1 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.

A-3-2 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.

A-3-3 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.

A-3-4 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.

A-3-5 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 B-4 “Device Error Codes” on page 377).

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.

A-3-6 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 4-3 "µC/FS File and Directory Names and Paths" on page 62 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 (such as `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.

A-3-7 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 5-4 “Partitions” on page 72.

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 B-4 “Device Error Codes” on page 377).

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.

A-3-8 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 5-4 "Partitions" on page 72.

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 B-4 "Device Error Codes" on page 377).

RETURNED VALUE

None.

NOTES/WARNINGS

Device state change will result from device I/O, not present or timeout error.

A-3-9 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 5-4 “Partitions” on page 72.

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 B-4 “Device Error Codes” on page 377).

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.

A-3-10 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 B-4 “Device Error Codes” on page 377).

RETURNED VALUE

None.

NOTES/WARNINGS

None.

A-3-11 FSDev_Rd()

```
void  FSDev_Rd (CPU_CHAR    *name_dev,
                void        *p_dest,
                FS_SEC_NBR   start,D
                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 B-4 “Device Error Codes” on page 377).

RETURNED VALUE

None.

NOTES/WARNINGS

Device state change will result from device I/O, not present or timeout error.

A-3-12 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 B-4 “Device Error Codes” on page 377).

RETURNED VALUE

DEF_YES, if the device has not 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.

A-3-13 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 B-4 “Device Error Codes” on page 377).

RETURNED VALUE

None.

NOTES/WARNINGS

Device state change will result from device I/O, not present or timeout error.

A-4 DIRECTORY ACCESS FUNCTIONS

```
void
FSDir_Close    (FS_DIR        *p_dir,
                FS_ERR        *p_err);
```

```
CPU_BOOLEAN
FSDir_IsOpen   (CPU_CHAR      *name_full,
                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);
```

A-4-1 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.

A-4-2 FSDir_IsOpen()

```
CPU_BOOLEAN  FSDir_Open (CPU_CHAR  *name_full,
                         FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dir.c	Application; fs_opendir(); FSEntry_*	FS_CFG_DIR_EN

Test if a directory is already open. This function is also called by various **FSEntry_*** functions to prevent concurrent access to an entry in the FAT filesystem.

ARGUMENTS

name_full Name of the directory. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

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_NAME_INVALID	Entry name specified invalid or volume could not be found.

Or entry error (see section B-8 "Entry Error Codes" on page 378).

RETURNED VALUE

DEF_NO, if dir is NOT open.

DEF_YES, if dir is open.

NOTES/WARNINGS

None.

A-4-3 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. See section 4-3 “µC/FS File and Directory Names and Paths” on page 62.

`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-8 “Entry Error Codes” on page 378).

RETURNED VALUE

Pointer to a directory, if NO errors.
Pointer to NULL, otherwise.

NOTES/WARNINGS

None.

A-4-4 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-5 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,
                   FS_FLAGS        entry_type,
                   CPU_BOOLEAN     excl,
                   FS_ERR         *p_err);
```

```
void
FSEntry_Del       (CPU_CHAR       *name_full,
                   FS_FLAGS        entry_type,
                   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);
```

A-5-1 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. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

`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 (See section B-8 "Entry Error Codes" on page 378).

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.

A-5-2 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. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

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 (See section B-8 “Entry Error Codes” on page 378).

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.

A-5-3 FSEntry_Create()

```
void   FSEntry_Create (CPU_CHAR     *name_full,
                       FS_FLAGS      entry_type,
                       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. See section 4-3 “µC/FS File and Directory Names and Paths” on page 62.

entry_type Indicates whether the new entry shall be a directory or a file (see Note #1) :

FS_ENTRY_TYPE_DIR, if the entry shall be a directory.
FS_ENTRY_TYPE_FILE, 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, entry_type is FS_ENTRY_TYPE_FILE and excl is `DEF_NO`, then the existing entry will be truncated. If the entry exists and is a directory and entry_type is FS_ENTRY_TYPE_DIR, 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.

A-5-4 FSEntry_Del()

```
void  FSEntry_Del (CPU_CHAR     *name_full,
                   FS_FLAGS      entry_type,
                   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. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

entry_type Indicates whether the entry MAY be a file (see Notes #1 and #2):

FS_ENTRY_TYPE_DIR,	if the entry must be a dir.
FS_ENTRY_TYPE_FILE,	if the entry must be a file.
FS_ENTRY_TYPE_ANY,	if the entry may be any type.

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.

A-5-5 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. See section 4-3 "μC/FS File and Directory Names and Paths" on page 62.

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.

A-5-6 FSEntry_Rename()

```
void  FSEntry_Rename (CPU_CHAR     *name_full_old,
                      CPU_CHAR     *name_full_new,
                      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. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

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.

A-5-7 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. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62.

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-6 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);
```

```
CPU_BOOLEAN
FSFile_IsOpen    (CPU_CHAR       *name_full,
                  FS_FLAGS       *p_mode,
                  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);
```

A-6-1 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.

A-6-2 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.

A-6-3 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.

A-6-4 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.

A-6-5 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.

A-6-6 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.

A-6-7 FSFile_IsOpen()

```
CPU_BOOLEAN  FSFile_IsOpen (CPU_CHAR  *name_full,
                            FS_FLAGS  *p_mode
                            FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_file.c	Application; FSFile_Open()	N/A

Test if file is already open.

ARGUMENTS

name_full Name of the file. See section 4-3 "µC/FS File and Directory Names and Paths" on page 62 for information about file names.

p_mode Pointer to variable that will receive the file access mode (see section 7-1-1 "Opening Files" on page 99 for the description the file access mode).

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_BUF_NONE_AVAIL	No buffer available.
FS_ERR_ENTRY_NOT_FILE	Entry NOT a file.
FS_ERR_NAME_INVALID	Invalid file name or path.
FS_ERR_VOL_INVALID_SEC_NBR	Invalid sector number found in directory entry.

RETURNED VALUE

DEF_NO if file is NOT open

DEF_YES if file is open.

NOTES/WARNINGS

None.

A-6-8 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.

A-6-9 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.

A-6-10 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.

A-6-11 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 4-3 “µC/FS File and Directory Names and Paths” on page 62 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.

- If `FS_FILE_ACCESS_MODE_TRUNC` is set, then `FS_FILE_ACCESS_MODE_WR` must also be set.
- If `FS_FILE_ACCESS_MODE_EXCL` is set, then `FS_FILE_ACCESS_MODE_CREATE` must also be set.
- `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 modes 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-1 **fs_fopen() mode strings and mode equivalents.**

A-6-12 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.

A-6-13 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.

A-6-14 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.

A-6-15 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.

A-6-16 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.

A-6-17 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-7 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);
```

A-7-1 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.

A-7-2 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`.

A-7-3 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.

A-7-4 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.

A-7-5 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.

A-7-6 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.

A-7-7 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.

`DEF_NO`, or the volume with name `name_vol` is not the default volume.

NOTES/WARNINGS

None.

A-7-8 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.

A-7-9 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..

A-7-10 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.

A-7-11 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 B-4 “Device Error Codes” on page 377).

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.

A-7-12 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.

A-7-13 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.

A-7-14 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-8 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);
```

A-8-1 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.

A-8-2 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.

A-8-3 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-9 NAND DRIVER FUNCTIONS

```
void
FSDev_NAND_LowFmt       (CPU_CHAR    *name_dev,
                         FS_ERR      *p_err);
```

```
void
FSDev_NAND_LowMount     (CPU_CHAR    *name_dev,
                         FS_ERR      *p_err);
```

```
void
FSDev_NAND_LowUnmount   (CPU_CHAR    *name_dev,
                         FS_ERR      *p_err);
```

```
void
FSDev_NAND_PhyRdSec     (CPU_CHAR    *name_dev,
                         void        *p_dest,
                         void        *p_spare
                         FS_SEC_NBR  sec_nbr_phy,
                         FS_ERR      *p_err);
```

```
void
FSDev_NAND_PhyWrSec     (CPU_CHAR    *name_dev,
                         void        *p_src,
                         void        *p_spare,
                         FS_SEC_NBR   sec_nbr_phy,
                         FS_ERR      *p_err);
```

```
void
FSDev_NAND_PhyEraseBlk (CPU_CHAR    *name_dev,
                         CPU_INT32U   blk_nbr_phy,
                         FS_ERR      *p_err);
```

A-9-1 FSDev_NAND_LowFmt()

```
void  FSDev_NAND_LowFmt (CPU_CHAR  *name_dev,
                         FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Low-level format a NAND device.

ARGUMENTS

name_dev Device name (see Note).

p_err Pointer to variable that will receive the 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 NAND device (e.g., “nand:0:”).

Low-level formating associates physical areas (sectors) of the device with logical sector numbers. A NAND 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.

A-9-2 FSDev_NAND_LowMount()

```
void  FSDev_NAND_LowMount (CPU_CHAR  *name_dev,
                           FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Low-level mount a NAND device.

ARGUMENTS

`name_dev` Device name (see Note).

`p_err` Pointer to variable that will receive the 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 NAND device (e.g., "`nand: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.

A-9-3 FSDev_NAND_LowUnmount()

```
void  FSDev_NAND_LowUnmount (CPU_CHAR  *name_dev,
                             FS_ERR    *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Low-level unmount a NAND 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 NAND device (e.g., “nand: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.

A-9-4 FSDev_NAND_PhyRdSec()

```
void  FSDev_NAND_PhyRdSec (CPU_CHAR    *name_dev,
                           void        *p_dest,
                           void        *p_spare,
                           FS_SEC_NBR   sec_nbr_phy,
                           FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Read sector from a NAND device and store data in buffer.

ARGUMENTS

name_dev Device name (see Note).

p_dest Pointer to destination buffer.

p_spare Pointer to buffer that will receive spare data.

sec_nbr_phy Sector to read.

p_err Pointer to variable that will receive the 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_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout.

RETURNED VALUE

None.

NOTES/WARNINGS

The device MUST be a NAND device (e.g., “`nand:0:`”).

A-9-5 FSDev_NAND_PhyWrSec()

```
void  FSDev_NAND_PhyWrSec (CPU_CHAR    *name_dev,
                           void        *p_src,
                           void        *p_spare
                           FS_SEC_NBR   sec_nbr_phy,
                           FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Write to a NAND device from a buffer.

ARGUMENTS

`name_dev` Device name (see Note).

`p_src` Pointer to source buffer.

`p_spare` Pointer to buffer that contains the spare data.

`sec_nbr_phy` Sector 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_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout.

RETURNED VALUE

None.

NOTES/WARNINGS

The device MUST be a NAND device (e.g., "`nand: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 page modified is NOT verified 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 page being programmed have not already been programmed.

A-9-6 FSDev_NAND_PhyEraseBlk()

```
void  FSDev_NAND_PhyEraseBlk (CPU_CHAR    *name_dev,
                              CPU_INT32U   blk_nbr_phy,
                              FS_ERR      *p_err);
```

File	Called from	Code enabled by
fs_dev_nand.c	Application	N/A

Erase block of NAND device.

ARGUMENTS

name_dev   Device name (see Note).

blk_nbr_phy   Block to erase.

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_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout.

RETURNED VALUE

None.

NOTES/WARNINGS

The device MUST be a NAND device (e.g., “nand: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 verified as being outside any existing file system or low-level format information.

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);
```

A-10-1 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.

A-10-2 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.

A-10-3 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.

A-10-4 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.

A-10-5 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.

A-10-6 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:`").

A-10-7 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_src 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.

A-10-8 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.

A-10-9 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);
```

A-11-1 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:`").

A-11-2 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.

A-11-3 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);
```

A-12-1 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.

A-12-2 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.

A-12-3 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.

A-12-4 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.

A-12-5 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.

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-1 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-2 BUFFER ERROR CODES

`FS_ERR_BUF_NONE_AVAIL`	No buffer available.

B-3 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-4 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-5 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-6 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-7 ECC ERROR CODES

FS_ERR_ECC_CORRECTABLE	Correctable ECC error.
FS_ERR_ECC_UNCORRECTABLE	Uncorrectable ECC error.

B-8 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.
FS_ERR_ENTRY_TYPE_INVALID	File system entry type is invalid.

FS_ERR_ENTRY_OPEN	Operation not allowed on entry corresponding to an open file/dir.

B-9 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.
FS_ERR_NAME_TOO_LONG	Full name is too long.

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.

B-15 OS LAYER ERROR CODES

FS_ERR_OS_LOCK	Lock not acquired.
FS_ERR_OS_INIT	OS not initialized.
FS_ERR_OS_INIT_LOCK	Lock signal not successfully initialized.
FS_ERR_OS_INIT_LOCK_NAME	Lock signal name not successfully initialized.

Appendix

C

µC/FS Porting Manual

µC/FS adapts to its environment via a number of ports. The simplest ones, common to all installations, interface with the application, OS kernel (if any) and CPU. More complicated may be ports to media drivers, which require additional testing, validation and optimization; but many of those are still straightforward. Figure C-1 diagrams the relationship between µC/FS and external modules and hardware.

The sections in this chapter describe each require function and give hints for implementers. Anyone creating a new port should first check the example ports are included in the µC/FS distribution in the following directory:

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

The port being contemplated may already exist; failing that, some similar CPU/device may have already be supported.

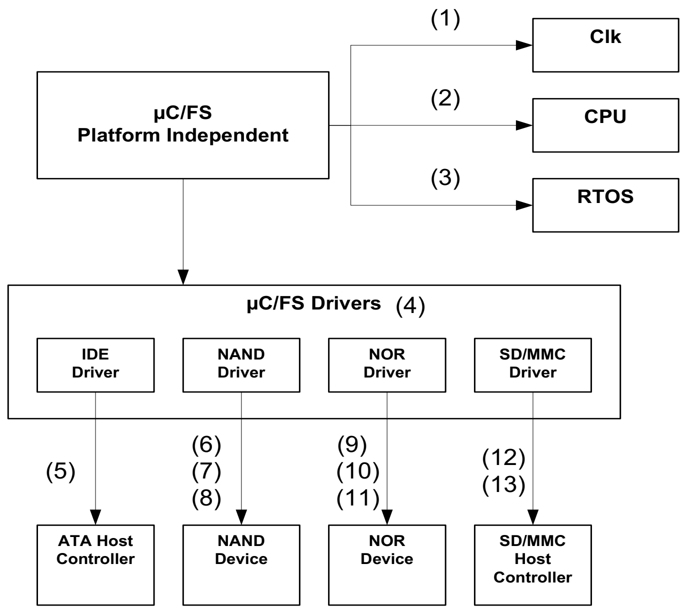


Figure C-1 **µC/FS Ports Architecture**

FC-1(1) µC/Clk act as a centralized clock management module. If you use an external real-time clock, you will have to write functions to let µC/FS know the date and time.

FC-1(2) The CPU port (within µC/CPU) adapts the file system suite to the CPU and compiler characteristics. The fixed-width types (e.g., `CPU_INT16U`) used in the file system suite are defined here.

FC-1(3) The RTOS port adapts the file system suite to the OS kernel (if any) included in the application. The files `FS_OS.C/H` contain functions primarily aimed at making accesses to devices and critical information in memory thread-safe.

FC-1(4) µC/FS interfaces with memory devices through drivers following a generic driver model. It is possible to create a driver for a different type of device from this model/template.

FC-1(5) The IDE/CF driver can be ported to any ATA host controller or bus interface.

FC-1(6) The NAND driver can be ported for many physical organizations (page size, bus width, etc.).

FC-1(7) The NAND driver can be ported to any bus interface. A NAND device can also be located directly on GPIO and accessed by direct toggling of port pins.

FC-1(8) The NAND driver can be ported to any SPI peripheral (for SPI flash). A NAND device can also be located directly on GPIO and accessed by direct toggling of port pins.

FC-1(9) The NOR driver can be ported to many physical organization (command set, bus type, etc.).

FC-1(10) The NOR driver can be ported to any bus interface.

FC-1(11) The NOR driver can be ported to any SPI peripheral (for SPI flash).

FC-1(12) The SD/MMC driver can be ported to any SD/MMC host controller for cardmode access.

FC-1(13) The SD/MMC driver can be ported to any SPI peripheral for SPI mode access.

C-1 DATE/TIME MANAGEMENT

Depending on the settings of μC/Clk, you might have to write time management functions that are specific to your application. For example, you might have to define the function **`Clk_ExtTS_Get()`** to obtain the timestamp of your system provided by a real-time clock peripheral. Please refer to μC/Clk manual for more details.

C-2 CPU PORT

μC/CPU is a processor/compiler port needed for μC/FS to be CPU/compiler-independant. Ports for the most popular architectures are already available in the μC/CPU distribution. If the μC/CPU port for your target architecture is not available, you should create your own based on the port template (also available in μC/CPU distribution). You should refer to the μC/CPU user manual to know how you should use it in your project.

C-3 OS KERNEL

μC/FS can be used with or without an RTOS. Either way, an OS port must be included in your project. The port includes one code/header file pair:

```
fs_os.c
fs_os.h
```

μC/FS manages devices and data structures that may not be accessed by severally tasks simultaneously. An OS kernel port leverages the kernel's mutual exclusion services (mutexes) for that purpose.

These files are generally placed in a directory named according to the following rubric:

```
\Micrium\Software\uC-FS\OS\<os_name>
```

Four sets of files are included with the μC/FS distribution:

`\Micrium\Software\uC-FS\OS\Template`	Template
`\Micrium\Software\uC-FS\OS\None`	No OS kernel port
`\Micrium\Software\uC-FS\OS\uCOS-II`	μC/OS-II port
`\Micrium\Software\uC-FS\OS\uCOS-III`	μC/OS-III port

If you don't use any OS (including a custom in-house OS), you should include the port for no OS in your project. You must also make sure that you manage interrupts correctly.

If you are using μC/OS-II or μC/OS-III, you should include the appropriate ports in your project. If you use another OS, you should create your own port based on the template. The functions that need to be written in this port are described here.

FS_OS_Init(), FS_OS_Lock() and FS_OS_Unlock()

The core data structures are protected by a single mutex. `FS_OS_Init()` creates this semaphore. `FS_OS_Lock()` and `FS_OS_Unlock()` acquire and release the resource. Lock operations are never nested.

FS_OS_DevInit(), FS_OS_DevLock() and FS_OS_DevUnlock()

File system device, generally, do not tolerate multiple simultaneous accesses. A different mutex controls access to each device and information about it in RAM. `FS_OS_DevInit()` creates one mutex for each possible device. `FS_OS_DevLock()` and `FS_OS_DevUnlock()` acquire and release access to a specific device. Lock operations for the same device are never nested.

FS_OS_FileInit(), FS_OS_FileAccept(), FS_OS_FileLock() and FS_OS_FileUnlock()

Multiple calls to file access functions may be required for a file operation that must be guaranteed atomic. For example, a file may be a conduit of data from one task to several. If a data entry cannot be read in a single file read, some lock is necessary to prevent preemption by another consumer. File locks, represented by API functions like `FSFile_LockGet()` and `flockfile()`, provide a solution. Four functions implement the actual lock in the OS port. `FS_OS_FileInit()` creates one mutex for each possible file. `FS_OS_FileLock()`/ `FS_OS_FileAccept()` and `FS_OS_FileUnlock()` acquire and release access to a specific file. Lock operations for the same file MAY be nested, so the implementations must be able to determine whether the active task owns the mutex. If it does, then an associated lock count should be incremented; otherwise, it should try to acquire the resource as normal.

`FS_OS_WorkingDirGet()` and **`FS_OS_WorkingDirSet()`**
File and directory paths are typically interpreted absolutely; they must start at the root directory, specifying every intermediate path component. If much work will be accomplished upon files in a certain directory or a task requires a root directory as part of its context, working directories are valuable. Once a working directory is set for a task, subsequent non-absolute paths will be interpreted relative to the set directory.

```
#if (FS_CFG_WORKING_DIR_EN == DEF_ENABLED)
CPU_CHAR  *FS_OS_WorkingDirGet (void)                                              (1)
{
    OS_ERR       os_err;
    CPU_INT32U   reg_val;
    CPU_CHAR    *p_working_dir;
    reg_val = OSTaskRegGet((OS_TCB *) 0,
                                      FS_OS_REG_ID_WORKING_DIR,
                                     &os_err);
    if (os_err != OS_ERR_NONE) {
        reg_val = 0u;
    }
    p_working_dir = (CPU_CHAR *)reg_val;
    return (p_working_dir);
}
#endif
#if (FS_CFG_WORKING_DIR_EN == DEF_ENABLED)
void  FS_OS_WorkingDirSet (CPU_CHAR  *p_working_dir)                               (2)
{
    OS_ERR      os_err;
    CPU_INT32U  reg_val;
    reg_val = (CPU_INT32U)p_working_dir;
    OSTaskRegSet((OS_TCB *) 0,
                            FS_OS_REG_ID_WORKING_DIR,
                            reg_val,
                           &os_err);
   (void)&os_err;
}
#endif
```

Listing C-1 **FS_OS_WorkingDirGet()/Set() (µC/OS-III)**

LC-1(1) `FS_OS_WorkingDirGet()` gets the pointer to the working directory associated with the active task. In µC/OS-III, the pointer is stored in one of the task registers, a set of software data that is part of the task context (just like hardware register values). The implantation casts the integral register value to a pointer to a character. If no working directory has been assigned, the return value must be a pointer to NULL. In the case of μC/OS-III, that will be done because the register values are cleared upon task creation.

LC-1(2) `FS_OS_WorkingDirSet()` associates a working directory with the active task. The pointer is cast to the integral register data type and stored in a task register.

The application calls either of the core file system functions `FS_WorkingDirSet()` or `fs_chdir()` to set the working directory. The core function forms the full path of the working directory and "saves" it with the OS port function `FS_OS_WorkingDirSet()`. The port function should associate it with the task in some manner so that it can be retrieved with `FS_OS_WorkingDirGet()` even after many context switches have occurred.

```
#if (FS_CFG_WORKING_DIR_EN == DEF_ENABLED)
void  FS_OS_WorkingDirFree (OS_TCB  *p_tcb)
{
    OS_ERR       os_err;
    CPU_INT32U   reg_val;
    CPU_CHAR    *path_buf;
    reg_val = OSTaskRegGet( p_tcb,
                            FS_OS_REG_ID_WORKING_DIR,
                           &os_err);
    if (os_err != OS_ERR_NONE) {
        return;
    }
    if (reg_val == 0u) {                                                    (1)
        return;
    }
    path_buf = (CPU_CHAR *)reg_val;
    FS_WorkingDirObjFree(path_buf);                                         (2)
}
#endif
```

Listing C-2 **FS_OS_WorkingDirFree() (μC/OS-III)**

LC-2(1) If the register value is zero, no working directory has been assigned and no action need be taken.

LC-2(2) `FS_WorkingDirObjFree()` frees the working directory object to the working directory pool. If this were not done, the unfreed object would constitute a memory leak that could deplete the heap memory eventually.

The character string for the working directory is allocated from the μC/LIB heap. If a task is deleted, it must be freed (made available for future allocation) to avert a crippling memory leak. The internal file system function `FS_WorkingDirObjFree()` releases the string to an object pool. In the port for μC/OS-III, that function is called by `FS_OS_WorkingDirFree()` which must be called by the assigned task delete hook.

FS_OS_Dly_ms()

Device drivers and example device driver ports delay task execution `FS_OS_Dly_ms()`. Common functions allow BSP developers to optimize implementation easily. A millisecond delay may be accomplished with an OS kernel service, if available. The trivial implementation of a delay (particularly a sub-millisecond delay) is a while loop; better performance may be achieved with hardware timers with semaphores for wait and asynchronous notification. The best solution will vary from one platform to another, since the additional context switches may prove burdensome. No matter which strategy is selected, the function MUST delay for at least the specified time amount; otherwise, sporadic errors can occur. Ideally, the actual time delay will equal the specified time amount to avoid wasting processor cycles.

```
void  FS_BSP_Dly_ms (CPU_INT16U  ms)
{
    /* $$$$ Insert code to delay for specified number of millieconds. */
}
```

Listing C-3 **FS_OS_Dly_ms()**

FS_OS_Sem####()

The four generic OS semaphore functions provide a complete abstraction of a basic OS kernel service. **FS_OS_SemCreate()** creates a semaphore which may later be deleted with **FS_OS_SemDel()**. **FS_OS_SemPost()** signals the semaphore (with or without timeout) and **FS_OS_SemPend()** waits until the semaphore is signaled. On systems without an OS kernel, the trivial implementations in Listing C-4 are recommended.

```
CPU_BOOLEAN  FS_OS_SemCreate (FS_BSP_SEM  *p_sem,                              (1)
                              CPU_INT16U   cnt)
{
   *p_sem = cnt;                    /* $$$$ Create semaphore with initial count 'cnt'. */
    return (DEF_OK);
}
CPU_BOOLEAN  FS_OS_SemDel (FS_BSP_SEM  *p_sem)                                 (2)
{
   *p_sem = 0u;                    /* $$$$ Delete semaphore. */
    return (DEF_OK);
}
```

Listing C-4 **FS_OS_Sem####() trivial implementations**

```
CPU_BOOLEAN  FS_OS_SemPend (FS_BSP_SEM  *p_sem,                                    (3)
                            CPU_INT32U   timeout)
{
    CPU_INT32U  timeout_cnts;
    CPU_INT16U  sem_val;
    CPU_SR_ALLOC();
    if (timeout == 0u) {
        sem_val = 0u;
        while (sem_val == 0u) {
            CPU_CRITICAL_ENTER();
            sem_val = *p_sem;           /* $$$$ If semaphore available ...               */
            if (sem_val > 0u) {
               *p_sem = sem_val - 1u;   /*      ... decrement semaphore count.           */
            }
            CPU_CRITICAL_EXIT();
        }
    } else {
        timeout_cnts = timeout * FS_BSP_CNTS_PER_MS;
        sem_val      = 0;
        while ((timeout_cnts >  0u) &&
               (sem_val      == 0u)) {
            CPU_CRITICAL_ENTER();
            sem_val = *p_sem;           /* $$$$ If semaphore available ...               */
            if (sem_val > 0) {
               *p_sem = sem_val - 1u;   /*      ... decrement semaphore count.           */
            }
            CPU_CRITICAL_EXIT();
            timeout_cnts--;
        }
    }
    if (sem_val == 0u) {
        return (DEF_FAIL);
    } else {
        return (DEF_OK);
    }
}
```

Listing C-5 **FS_OS_Sem####() trivial implementations (continued)**

LC-5(1) `FS_OS_SemCreate()` creates a semaphore in the variable `p_sem`. For this trivial implementation, `FS_BSP_SEM` is a integer type which stores the current count, i.e., the number of objects available.

LC-5(2) `FS_OS_SemDel()` deletes a semaphore created by `FS_OS_SemCreate()`.

```
CPU_BOOLEAN  FS_OS_SemPost (FS_BSP_SEM  *p_sem)                                        (4)
{
    CPU_INT16U  sem_val;
    CPU_SR_ALLOC();
    CPU_CRITICAL_ENTER();
    sem_val = *p_sem;               /* $$$$ Increment semaphore value. */
    sem_val++;
   *p_sem   =  sem_val;
    CPU_CRITICAL_EXIT();
    return (DEF_OK);
}
```

Listing C-6 **FS_OS_Sem####() trivial implementations (continued)**

LC-6(3) **`FS_OS_SemPend()`** waits until a semaphore is signaled. If a zero timeout is given, the wait is possibly infinite (it never times out).

LC-6(4) **`FS_OS_SemPost()`** signals a semaphore.

C-4 DEVICE DRIVER

Devices drivers for the most popular devices are already available for µC/FS. If you use a particular device for which no driver exist, you should read this section to understand how to build your own 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, `FS_DEV_API`, includes pointers to eight functions used to control and access the device:

```
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
};
```

The functions which must be implemented are listed and described in Table C-1. The first two functions, `NameGet()` and `Init()`, act upon the driver as a whole; neither should interact with any physical devices. The remaining functions act upon individual devices, and the first argument of each is a pointer to a `FS_DEV` structure which holds device information, including the unit number which uniquely identifies the device unit (member `UnitNbr`).

Function	**Description**
`NameGet()`	Get driver name.
`Init()`	Initialize driver.
`Open()`	Open a device.
`Close()`	Close a device.
`Rd()`	Read from a device.
`Wr()`	Write to a device.
`Query()`	Get information about a device.

Function	Description
`IO_Ctrl()`	Execute device I/O control operation.

Table C-1 **Device Driver API Functions**

C-4-1 NameGet()

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

File	Called from	Code enabled by
fs_dev_####.c	various	N/A

Device drivers are identified by unique names, on which device names are based. For example, the unique name for the NAND flash driver is "nand"; the NAND devices will be named "nand:0:", "nand:1:", etc.

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.

C-4-2 Init()

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

File	Called from	Code enabled by
fs_dev_####.c	FS_DevDrvAdd()	N/A

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.

ARGUMENTS

None.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-4-3 Open()

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

File	Called from	Code enabled by
fs_dev_####.c	FSDev_Open()	N/A

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.

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.

C-4-4 Close()

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

File	Called from	Code enabled by
fs_dev_####.c	FSDev_Close()	N/A

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

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.

C-4-5 Rd()

```
static  void  FSDev_####_Rd (FS_DEV       *p_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	FSDev_RdLocked()	N/A

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.

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.

C-4-6 Wr()

```
static  void  FSDev_####_Wr (FS_DEV       *p_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`	`FSDev_WrLocked()`	N/A

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.

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.

C-4-7 Query()

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

File	Called from	Code enabled by
fs_dev_####.c	FSDev_Open(), FSDev_Refresh(), FSDev_QueryLocked()	N/A

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

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 D-2 "FS_DEV_INFO" on page 510.

C-4-8 IO_Ctrl()

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

File	Called from	Code enabled by
fs_dev_####.c	various	N/A

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

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.

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

C-5 IDE/CF DEVICE BSP

If you use and IDE/CF device, a driver is already available for μC/FS. A BSP is required so that the IDE driver will work on a particular system. The port includes one code file:

`FS_DEV_IDE_BSP.C`

Several example ports are included in the μC/FS distribution in files named according to the following rubric:

`\Micrium\Software\uC-FS\Examples\BSP\Dev\IDE\<manufacturer>\<board name>`

Each BSP must implement the functions in Table C-2.

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 C-2 **IDE/CF BSP Functions**

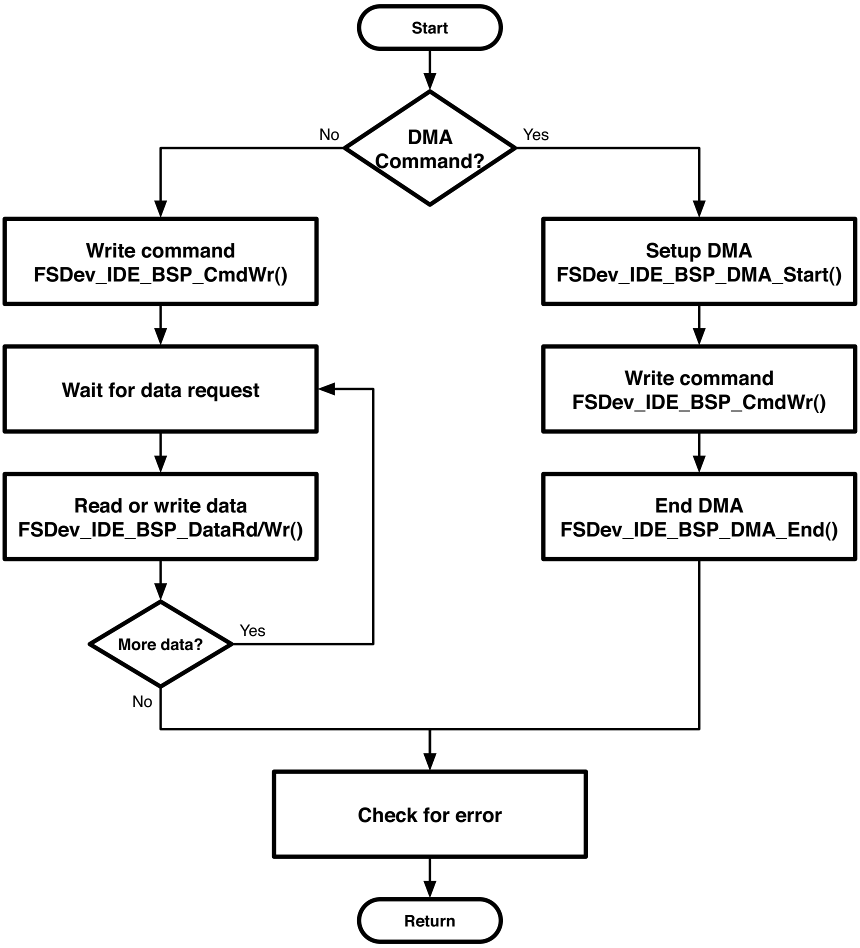


Figure C-2 **Command Execution**

C-5-1 FSDev_IDE_BSP_Open()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_Refresh()	N/A

Initialize IDE/CF hardware.

ARGUMENTS

unit_nbr Unit number of IDE/CF device.

RETURNED VALUE

DEF_OK, if interface was opened

DEF_FAIL, otherwise

NOTES/WARNINGS

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

C-5-2 FSDev_IDE_BSP_Close()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_Close()	N/A

Uninitialize IDE/CF hardware.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF device.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-5-3 FSDev_IDE_BSP_Lock() / FSDev_IDE_BSP_Unlock()

```
void  FSDev_IDE_BSP_Lock   (FS_QTY  unit_nbr);
void  FSDev_IDE_BSP_Unlock (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_ide_bsp.c`	various	N/A

Acquire/release IDE/CF bus lock.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF device.

RETURNED VALUE

None.

NOTES/WARNINGS

`FSDev_IDE_BSP_Lock()` will be called before the IDE/CF driver begins to access the IDE/CF 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.

C-5-4 FSDev_IDE_BSP_Reset()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_Refresh()	N/A

Hardware-reset the IDE/CF device.

ARGUMENTS

unit_nbr   Unit number of IDE/CF device.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-5-5 FSDev_IDE_BSP_RegRd()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	various	N/A

Read from IDE/CF device register.

ARGUMENTS

unit_nbr Unit number of IDE/CF 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.

C-5-6 FSDev_IDE_BSP_RegWr()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	various	N/A

Write to IDE/CF device register.

ARGUMENTS

unit_nbr Unit number of IDE/CF device.

reg Register to read:

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 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_DEVCTRL	Device Control Register.

val Value to write into register.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-5-7 FSDev_IDE_BSP_CmdWr()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_RdData() FSDev_IDE_WrData()	N/A

Write 7-byte command to IDE/CF device registers.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF 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  = cmd[0]
REG_SC  = cmd[1]
REG_SN  = cmd[2]
REG_CYL = cmd[3]
REG_CYN = cmd[4]
REG_DH  = cmd[5]
REG_CMD = cmd[6]
```

C-5-8 FSDev_IDE_BSP_DataRd()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_RdData()	N/A

Read data from IDE/CF device.

ARGUMENTS

unit_nbr Unit number of IDE/CF device.

p_dest Pointer to destination memory buffer.

cnt Number of octets of data to read.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-5-9 FSDev_IDE_BSP_DataWr()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_WrData()	N/A

Write data to IDE/CF device.

ARGUMENTS

unit_nbr Unit number of IDE/CF device.

p_src Pointer to source memory buffer.

cnt Number of octets of data to write.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-5-10 FSDev_IDE_BSP_DMA_Start()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_RdData() FSDev_IDE_WrData()	N/A

Setup DMA for command (initialize channel).

ARGUMENTS

`unit_nbr` Unit number of IDE/CF device.

`p_data` 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

DMA setup occurs before the command is executed (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.

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.

C-5-11 FSDev_IDE_BSP_DMA_End()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_RdData() FSDev_IDE_WrData()	N/A

Setup DMA for command (initialize channel).

ARGUMENTS

unit_nbr Unit number of IDE/CF device.

p_data Pointer to memory buffer.

cnt Number of octets to transfer.

rd Indicates whether transfer was read or write:

DEF_YES	Transfer was read.
DEF_NO	Transfer was write.

RETURNED VALUE

None.

NOTES/WARNINGS

DMA setup occurs before the command is executed (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.

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

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.

C-5-12 FSDev_IDE_BSP_GetDrvNbr()

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

File	Called from	Code enabled by
`fs_dev_ide_bsp.c`	`FSDev_IDE_Refresh()`	N/A

Get IDE/CF driver number.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF 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.

C-5-13 FSDev_IDE_BSP_GetModesSupported()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_Refresh()	N/A

Get supported transfer modes.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF device.

RETURNED VALUE

Bit-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.

C-5-14 FSDev_IDE_BSP_SetMode()

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

File	Called from	Code enabled by
fs_dev_ide_bsp.c	FSDev_IDE_Refresh()	N/A

Set transfer mode timings.

ARGUMENTS

unit_nbr Unit number of IDE/CF 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.

C-5-15 FSDev_IDE_BSP_Dly400_ns()

CPU_INT08U FSDev_IDE_BSP_Dly400_ns (FS_QTY unit_nbr);

File	Called from	Code enabled by
`fs_dev_ide_bsp.c`	`various`	N/A

Delay 400-ns.

ARGUMENTS

`unit_nbr` Unit number of IDE/CF device.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6 NAND Flash Physical-Layer Driver

The NAND driver is divided into three layers. The topmost layer, the generic driver, requires an intermediate physical-layer driver to effect flash operations like erasing blocks and writing octets depending on the memory type and organization. The physical-layer driver is already available for different architectures and includes one code/header file pair named according to the following rubric:

```
FS_DEV_NAND_<device_name>.C
FS_DEV_NAND_<device_name>.H
```

The physical-layer driver acts via a BSP. The generic drivers for traditional NAND flash require a BSP as described in Appendix C, "NAND Flash BSP" on page 440. The drivers for SPI flash (like Atmel Dataflash) require a SPI BSP as described in Appendix C, "NAND Flash SPI BSP" on page 450.

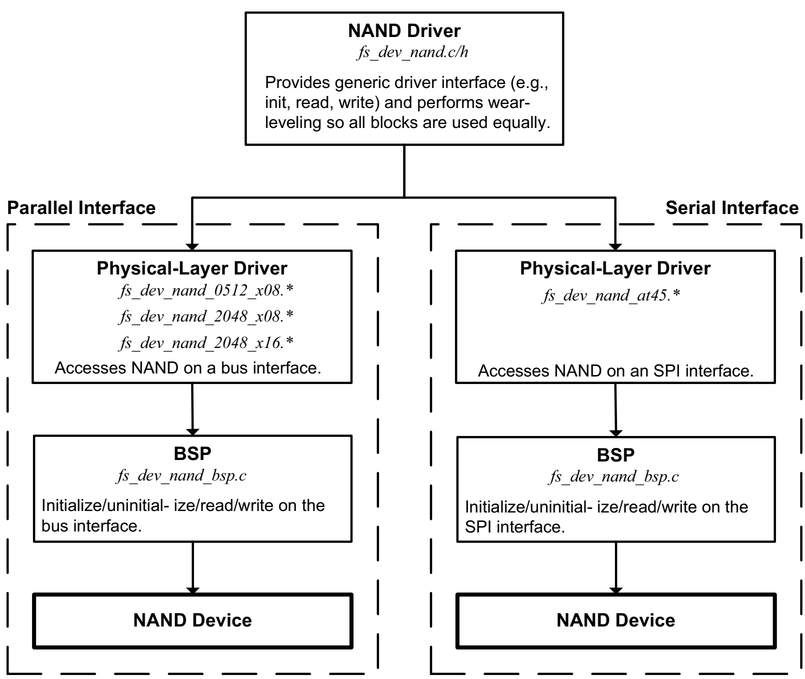


Figure C-3 **NAND Driver Architecture**

Each physical-layer driver must implement the functions to be placed into a `FS_DEV_NAND_PHY_API` structure:

```
const FS_DEV_NAND_PHY_API  FSDev_NAND_#### {
    FSDev_NAND_PHY_Open,
    FSDev_NAND_PHY_Close,
    FSDev_NAND_PHY_RdPage,
    FSDev_NAND_PHY_RdSpare,
    FSDev_NAND_PHY_WrPage,
    FSDev_NAND_PHY_WrSpare,
    FSDev_NAND_PHY_CopyBack,
    FSDev_NAND_PHY_EraseBlk,
    FSDev_NAND_PHY_IO_Ctrl,
};
```

The functions which must be implemented are listed and described in Table C-5. The first argument of each of these is a pointer to a `FS_DEV_NAND_PHY_DATA` structure which holds physical device information. Specific members will be described in subsequent sections as necessary. The NAND driver populates an internal instance of this type based upon configuration information. Before the file system suite has been initialized, the application may do the same if raw device accesses are a necessary part of its start-up procedure.

Function	**Description**
`Open()`	Open (initialize) a NAND device and get NAND device information.
`Close()`	Close (uninitialize) a NAND device.
`RdPage()`	Read a page from a NAND device and store data in buffer.
`RdSpare()`	Read a spare area from a NAND device and store data in buffer.
`WrPage()`	Write to a page of a NAND device from data in buffer.
`WrSpare()`	Write to a spare area of a NAND device from data in buffer.
`WrCopyBack)`	Copy data from one block to another.
`EraseBlk()`	Erase block of NAND device.
`IO_Ctrl()`	Perform NAND device I/O control operation.

Table C-3 **NAND flash physical-layer driver functions**

C-6-1 Open()

```
void  Open (FS_DEV_NAND_PHY_DATA  *p_phy_data,
            FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	`FSDev_NAND_Open()`	N/A

Open (initialize) a NAND device instance and get NAND device information.

ARGUMENTS

`p_phy_data` Pointer to NAND phy data.

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

RETURNED VALUE

None.

NOTES/WARNINGS

Several members of `p_phy_data` may need to be used/assigned:

1 `UnitNbr` is the unit number of the NAND device.

2 `BlkCnt` and `BlkSize` MUST be assigned the block count and block size of the device, respectively. A block is the device erase unit, e.g., the smallest area of the device that can be erased at any time.

3 `PageSize MUST` be assigned the page size of the device. A page is the device program unit, i.e., the smallest area of the device that can be programmed at any time.

4 `BlkSize` MUST be a multiple of `PageSize`.

5 `PageSize` MUST be a multiple of `SecSize`.

6 `SpareSize` MUST be assigned the size (in bytes) of the spare arear per sector.

7 **`MaxClkFreq`** specifies the maximum SPI clock frequency.

8 **`BusWidth`** specify the bus configuration.

C-6-2 Close()

```
void  Close (FS_DEV_NAND_PHY_DATA  *p_phy_data);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_Close()	N/A

Close (uninitialize) a NAND device instance.

ARGUMENTS

p_phy_data Pointer to NAND phy data.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6-3 RdPage()

```
void  RdPage (FS_DEV_NAND_PHY_DATA  *p_phy_data,
              void                  *p_dest,
              void                  *p_dest_spare,
              FS_SEC_NBR             sec_nbr_phy,
              FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_PhyRdSecHandler()	N/A

Read from a NAND device and store data in buffer.

ARGUMENTS

p_phy_data Pointer to NAND phy data.

p_dest Pointer to destination buffer.

p_dest_spare Pointer to destination spare buffer.

sec_nbr_phy Physical sector to read from the page.

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

FS_ERR_NONE	Octets read successfully.
FS_ERR_DEV_INVALID_OP	Device invalid operation.
FS_ERR_DEV_INVALID_ECC	Invalid ECC.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6-4 RdSpare()

```
void  RdSpare (FS_DEV_NAND_PHY_DATA  *p_phy_data,
               void                  *p_dest,
               FS_SEC_NBR             sec_nbr_phy,
               CPU_INT08U             offset,
               CPU_INT08U             bytes_nbr,
               FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_PhyRdSpareHandler()	N/A

Read data from NAND page spare area and store data in buffer.

ARGUMENTS

`p_phy_data` Pointer to NAND phy data.

`p_dest` Pointer to destination buffer.

`sec_nbr_phy` Physical sector to read from the page.

`offset` Offset in the spare area.

`bytes_nbr` Number of bytes to read.

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

FS_ERR_NONE	Octets read successfully.
FS_ERR_DEV_INVALID_OP	Device invalid operation.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

C-6-5 WrPage()

```
void  WrPage (FS_DEV_NAND_PHY_DATA  *p_phy_data,
              void                  *p_src,
              void                  *p_src_spare,
              FS_SEC_NBR             sec_nbr_phy,
              FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_PhyWrSecHandler()	N/A

Write to a NAND device.

ARGUMENTS

p_phy_data Pointer to NAND phy data.

p_src Pointer to source buffer.

p_src_spare Pointer to source spare buffer.

sec_nbr_phy Physical sector to write.

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

FS_ERR_NONE	Octets written successfully.
FS_ERR_DEV_INVALID_OP	Device invalid operation.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6-6 WrSpare()

```
void  WrSpare (FS_DEV_NAND_PHY_DATA  *p_phy_data,
               void                  *p_src,
               FS_SEC_NBR             sec_nbr_phy,
               CPU_INT08U             offset,
               CPU_INT08U             bytes_nbr,
               FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_PhyWrSpareHandler()	N/A

Write data to a NAND device page spare area.

ARGUMENTS

p_phy_data Pointer to NAND phy data.

p_src Pointer to source buffer.

sec_nbr_phy Sector number for which the spare area will be written.

offset Offset in the spare area.

bytes_nbr Number of bytes to write.

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

FS_ERR_NONE	Octets written successfully.
FS_ERR_DEV_INVALID_OP	Device invalid operation.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

C-6-7 CopyBack()

```
void  CopyBack (FS_DEV_NAND_PHY_DATA  *p_phy_data,
                CPU_INT32U             src_page_nbr_phy,
                CPU_INT32U             dest_page_nbr_phy,
                FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	N/A	N/A

Make internal copy back of page data.

ARGUMENTS

`p_phy_data` Pointer to NAND phy data.

`src_page_nbr_phy` Source page number.

`dest_page_nbr_phy` Destination page number.

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

`FS_ERR_NONE` Page copied successfully.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6-8 EraseBlk()

```
void  EraseBlk (FS_DEV_NAND_PHY_DATA  *p_phy_data,
                CPU_INT32U             blk_nbr_phy,
                FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	FSDev_NAND_PhyEraseBlkHandler()	N/A

Erase block of NAND device.

ARGUMENTS

p_phy_data Pointer to NAND phy data.

blk_nbr_phy Block to erase.

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

FS_ERR_NONE	Block erased successfully.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-6-9 IO_Ctrl()

```
void  IO_Ctrl (FS_DEV_NAND_PHY_DATA  *p_phy_data,
               CPU_INT08U             opt,
               void                  *p_data,
               FS_ERR                *p_err);
```

File	Called from	Code enabled by
NAND physical-layer driver	N/A	N/A

Perform NAND device I/O control operation.

ARGUMENTS

`p_phy_data` Pointer to NAND phy data.

`opt` Control command.

`p_data` Buffer which holds data to be used for operation.
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_DEV_INVALID_IO_CTRL` `I/O` control unknown to driver.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

Table C-4

C-7 NAND Flash BSP

The NAND driver must adapt to the specific hardware using a BSP. The following functions must be implemented to interface the NAND driver on a parallel bus.

C-7-1 FSDev_NAND_BSP_Open()

```
CPU_BOOLEAN  FSDev_NAND_BSP_Open (FS_QTY      unit_nbr,
                                  CPU_INT08U  bus_width);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Open (initialize) bus for NAND.

ARGUMENTS

`unit_nbr` Unit number of NAND.

`bus_width` Bus width, in bits.

RETURNED VALUE

`DEF_OK`, if interface was opened.

`DEF_FAIL`, otherwise.

NOTES/WARNINGS

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

C-7-2 FSDev_NAND_BSP_Close()

```
void   FSDev_NAND_BSP_Close (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Close (uninitialize) bus for NAND.

ARGUMENTS

`unit_nbr` Unit number of NAND.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-7-3 FSDev_NAND_BSP_ChipSelEn()

```
void  FSDev_NAND_BSP_ChipSelEn (FS_QTY       unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_nand_bsp.c`	NAND physical-layer driver	N/A

Enable NAND chip select / chip enable.

ARGUMENTS

`unit_nbr` Unit number of NAND.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-4 FSDev_NAND_BSP_ChipSelDis()

```
void  FSDev_NAND_BSP_ChipSelDis (FS_QTY       unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_nand_bsp.c`	NAND physical-layer driver	N/A

Disable NAND chip select / chip enable.

ARGUMENTS

`unit_nbr` Unit number of NAND.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-5 FSDev_NAND_BSP_RdData()

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

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Read data from NAND.

ARGUMENTS

`unit_nbr` Unit number of NAND.

`p_dest` Pointer destination memory buffer.

`cnt` Number of octets to read.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-6 FSDev_NAND_BSP_WrAddr()

```
void  FSDev_NAND_BSP_WrAddr (FS_QTY      unit_nbr,
                             CPU_INT08U *p_addr);
                             CPU_SIZE_T  cnt);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Write address to NAND.

ARGUMENTS

unit_nbr Unit number of NAND.

p_addr Pointer to buffer that holds address.

cnt Number of octets to write.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-7 FSDev_NAND_BSP_WrCmd()

```
void  FSDev_NAND_BSP_WrCmd  (FS_QTY      unit_nbr,
                             CPU_INT08U *p_cmd);
                             CPU_SIZE_T  cnt);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Write command to NAND.

ARGUMENTS

`unit_nbr` Unit number of NAND.

`p_cmd` Pointer to buffer that holds command.

`cnt` Number of octets to write.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-8 FSDev_NAND_BSP_WrData()

```
void  FSDev_NAND_BSP_WrData (FS_QTY      unit_nbr,
                             CPU_INT08U *p_src);
                             CPU_SIZE_T  cnt);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Write data to NAND.

ARGUMENTS

unit_nbr Unit number of NAND.

p_src Pointer to source memory buffer

cnt Number of octets to write.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-7-9 FSDev_NAND_BSP_WaitWhileBusy()

```
CPU_BOOLEAN  FSDev_NAND_BSP_WaitWhileBusy

        (FS_QTY                unit_nbr,
         FS_DEV_NAND_PHY_DATA  *p_addr,
         CPU_BOOLEAN          (*poll_fnct)(FS_DEV_NAND_PHY_DATA *),
         CPU_INT32U             to_us);
```

File	Called from	Code enabled by
fs_dev_nand_bsp.c	NAND physical-layer driver	N/A

Wait while NAND is busy.

ARGUMENTS

unit_nbr Unit number of NAND.

p_phy_data Pointer to NAND 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 NAND became ready.

DEF_FAIL, otherwise.

NOTES/WARNINGS

None.

C-8 NAND Flash SPI BSP

The NAND driver must adapt to the specific hardware using a BSP. A serial NAND Flash will be interfaced on a SPI bus. See Appendix C, “SPI BSP” on page 494 for the details on how to implement the software port for your SPI bus.

C-9 NOR Flash Physical-Layer Driver

The NOR driver is divided into three layers. The topmost layer, the generic driver, requires an intermediate physical-layer driver to effect flash operations like erasing blocks and writing octets. The physical-layer driver includes one code/header file pair named according to the following rubric:

```
FS_DEV_NOR_<device_name>.C
FS_DEV_NOR_<device_name>.H
```

A non-uniform flash—a flash with some blocks of one size and some blocks of another—will require a custom driver adapted from the generic driver for the most similar medium type. Multiple small blocks should be grouped together to form large blocks, effectively making the flash appear uniform to the generic driver. A custom physical-layer driver can also implement advanced program operations unique to a NOR device family.

The physical-layer driver acts via a BSP. The generic drivers for traditional NOR flash require a BSP as described in Appendix C, “NOR Flash BSP” on page 459. The drivers for SPI flash require a SPI BSP as described in Appendix C, “NOR Flash SPI BSP” on page 466.

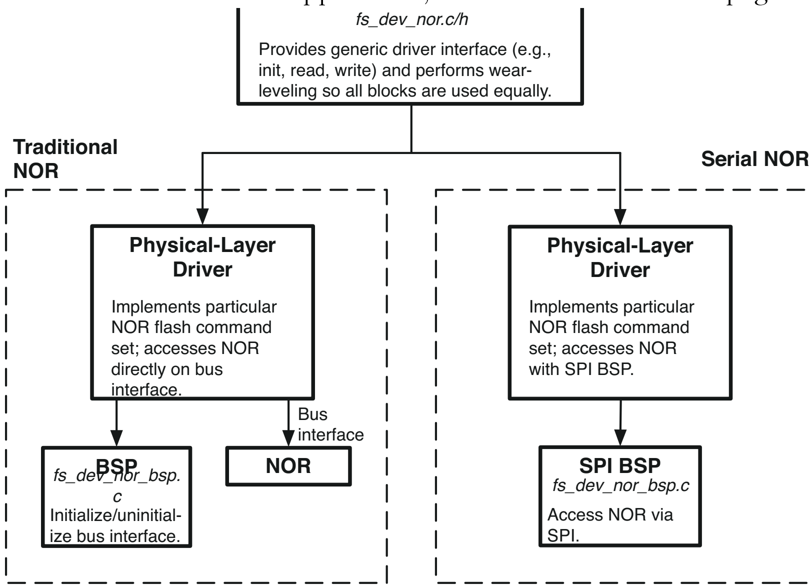


Figure C-4 **NOR Driver Architecture**

Each physical-layer driver must implement the functions to be placed into a **`FS_DEV_NOR_PHY_API`** structure:

```
const FS_DEV_NOR_PHY_API  FSDev_NOR_#### {
    FSDev_NOR_PHY_Open,
    FSDev_NOR_PHY_Close,
    FSDev_NOR_PHY_Rd,
    FSDev_NOR_PHY_Wr,
    FSDev_NOR_PHY_EraseBlk,
    FSDev_NOR_PHY_IO_Ctrl,
};
```

The functions which must be implemented are listed and described in Table C-5. The first argument of each of these is a pointer to a **`FS_DEV_NOR_PHY_DATA`** structure which holds physical device information. Specific members will be described in subsequent sections as necessary. The NOR driver populates an internal instance of this type based upon configuration information. Before the file system suite has been initialized, the application may do the same if raw device accesses are a necessary part of its start-up procedure.

Function	Description
`Open()`	Open (initialize) a NOR device and get NOR device information.
`Close()`	Close (uninitialize) a NOR device.
`Rd()`	Read from a NOR device and store data in buffer.
`Wr()`	Write to a NOR device from a buffer.
`EraseBlk()`	Erase block of NOR device.
`IO_Ctrl()`	Perform NOR device I/O control operation.

Table C-5 **NOR flash physical-layer driver functions**

C-9-1 Open()

```
void  Open (FS_DEV_NOR_PHY_DATA  *p_phy_data,
            FS_ERR               *p_err);
```

File	Called from	Code enabled by
NOR physical-layer driver	`FSDev_NOR_Open()`	N/A

Open (initialize) a NOR device instance and get NOR device information.

ARGUMENTS

`p_phy_data` Pointer to NOR phy data.

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

RETURNED VALUE

None.

NOTES/WARNINGS

Several members of `p_phy_data` may need to be used/assigned:

1 `BlkCnt` and `BlkSize` MUST be assigned the block count and block size of the device, respectively.

2 `RegionNbr` specifies the block region that will be used. `AddrRegionStart` MUST be assigned the start address of this block region.

3 `DataPtr` may store a pointer to any driver-specific data.

4 `UnitNbr` is the unit number of the NOR device.

5 `MaxClkFreq` specifies the maximum SPI clock frequency.

6 BusWIdth, BusWidthMax and PhyDevCnt specify the bus configuration. AddrBase specifies the base address of the NOR flash memory.

C-9-2 Close()

```
void  Close (FS_DEV_NOR_PHY_DATA  *p_phy_data);
```

File	Called from	Code enabled by
NOR physical-layer driver	FSDev_NOR_Close()	N/A

Close (uninitialize) a NOR device instance.

ARGUMENTS

`p_phy_data` Pointer to NOR phy data.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-9-3 Rd()

```
void  Rd (FS_DEV_NOR_PHY_DATA  *p_phy_data,
          void                 *p_dest,
          CPU_INT32U            start,
          CPU_INT32U            cnt,
          FS_ERR               *p_err);
```

File	Called from	Code enabled by
NOR physical-layer driver	FSDev_NOR_PhyRdHandler()	N/A

Read from a NOR device and store data in buffer.

ARGUMENTS

p_phy_data Pointer to NOR phy data.

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 receive the return error code from this function.

FS_ERR_NONE	Octets read successfully.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-9-4 Wr()

```
void  Wr (FS_DEV_NOR_PHY_DATA  *p_phy_data,
          void                 *p_src,
          CPU_INT32U            start,
          CPU_INT32U            cnt,
          FS_ERR               *p_err);
```

File	Called from	Code enabled by
NOR physical-layer driver	FSDev_NOR_PhyWrHandler()	N/A

Write to a NOR device from a buffer.

ARGUMENTS

p_phy_data Pointer to NOR phy data.

p_src 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 receive the return error code from this function.

FS_ERR_NONE	Octets written successfully.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-9-5 EraseBlk()

```
void  EraseBlk (FS_DEV_NOR_PHY_DATA  *p_phy_data,
                CPU_INT32U            start,
                CPU_INT32U            size,
                FS_ERR               *p_err);
```

File	Called from	Code enabled by
NOR physical-layer driver	FSDev_NOR_PhyEraseBlkHandler()	N/A

Erase block of NOR device.

ARGUMENTS

p_phy_data Pointer to NOR phy data.

start Start address of block (relative to start of device).

size Size of block, in octets

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

FS_ERR_NONE	Block erased successfully.
FS_ERR_DEV_INVALID_OP	Invalid operation for device.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-9-6 IO_Ctrl()

```
void  IO_Ctrl (FS_DEV_NOR_PHY_DATA  *p_phy_data,
               CPU_INT08U            opt,
               void                 *p_data,
               FS_ERR               *p_err);
```

File	Called from	Code enabled by
NOR physical-layer driver	various	N/A

Perform NOR device I/O control operation.

ARGUMENTS

`p_phy_data` Pointer to NOR phy data.

`opt` Control command.

`p_data` Buffer which holds data to be used for operation.
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 unknown to driver.
FS_ERR_DEV_INVALID_OP	Invalid operation for device.
FS_ERR_DEV_IO	Device I/O error.
FS_ERR_DEV_TIMEOUT	Device timeout error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-10 NOR Flash BSP

A “traditional” NOR flash has two buses, one for addresses and another for data. For example, the host initiates a data read operation with the address of the target location latched onto the address bus; the device responds by outputting a data word on the data bus.

A BSP abstracts the flash interface for the physical layer driver. The port includes one code file:

`FS_DEV_NOR_BSP.C`

This file is generally placed with other BSP files in a directory named according to the following rubric:

`\Micrium\Software\EvalBoards\<manufacturer>\<board_name>`
`\<compiler>\BSP\`

Function	Description
`FSDev_NOR_BSP_Open()`	Open (initialize) bus for NOR
`FSDev_NOR_BSP_Close()`	Close (uninitialize) bus for NOR.
`FSDev_NOR_BSP_Rd_08()/16()`	Read from bus interface.
`FSDev_NOR_BSP_RdWord_08()/16()`	Read word from bus interface.
`FSDev_NOR_BSP_WrWord_08()/16()`	Write word to bus interface.
`FSDev_NOR_BSP_WaitWhileBusy()`	Wait while NOR is busy.

Table C-6 **NOR BSP Functions**

C-10-1 FSDev_NOR_BSP_Open()

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

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Open (initialize) bus for NOR.

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.

C-10-2 FSDev_NOR_BSP_Close()

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

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Close (uninitialize) bus for NOR.

ARGUMENTS

`unit_nbr` Unit number of NOR.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-10-3 FSDev_NOR_BSP_Rd_XX()

```
void  FSDev_NAND_BSP_Rd_08 (FS_QTY        unit_nbr,
                            void         *p_dest,
                            CPU_ADDR      addr_src,
                            CPU_SIZE_T    cnt);
void  FSDev_NAND_BSP_Rd_16 (FS_QTY        unit_nbr,
                            void         *p_dest,
                            CPU_ADDR      addr_src,
                            CPU_SIZE_T    cnt);
```

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Read data from bus interface.

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 its access width will be called.

C-10-4 FSDev_NOR_BSP_RdWord_XX()

```
CPU_INT08U  FSDev_NAND_BSP_RdWord_08 (FS_QTY    unit_nbr,
                                      CPU_ADDR  addr_src);
CPU_INT16U  FSDev_NAND_BSP_RdWord_16 (FS_QTY    unit_nbr,
                                      CPU_ADDR  addr_src);
```

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Read data from bus interface.

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 its access width will be called.

C-10-5 FSDev_NOR_BSP_WrWord_XX()

```
void  FSDev_NAND_BSP_WrWord_08 (FS_QTY      unit_nbr,
                                CPU_ADDR    addr_src,
                                CPU_INT08U  datum);
void  FSDev_NAND_BSP_WrWord_16 (FS_QTY      unit_nbr,
                                CPU_ADDR    addr_src,
                                CPU_INT16U  datum);
```

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Write data to bus interface.

ARGUMENTS

unit_nbr Unit number of NOR.

addr_src Source address.

datum Word to write.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-10-6 FSDev_NOR_BSP_WaitWhileBusy()

```
CPU_BOOLEAN
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);
```

File	Called from	Code enabled by
fs_dev_nor_bsp.c	NOR physical-layer driver	N/A

Wait while NAND is busy.

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 NAND became ready.

`DEF_FAIL`, otherwise.

NOTES/WARNINGS

None.

```
CPU_BOOLEAN  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)
{
    CPU_INT32U   time_cur_us;
    CPU_INT32U   time_start_us;
    CPU_BOOLEAN  rdy;
    time_cur_us   = /* $$$$ GET CURRENT TIME, IN MICROSECONDS. */;
    time_start_us = time_cur_us;
    while (time_cur_us - time_start_us < to_us) {                                    (1)
        rdy = poll_fnct(p_phy_data);                                                 (2)
        if (rdy == DEF_OK) {
            return (DEF_OK);
        }
        time_cur_us = /* $$$$ GET CURRENT TIME, IN MICROSECONDS. */;
    }
    return (DEF_FAIL);                                                               (3)
}
```

Listing C-7 **FSDev_NOR_BSP_WaitWhileBusy() (without hardware read/busy signal)**

LC-7(1) At least to_us microseconds should elapse before the function gives up and returns. Returning early can cause disruptive timeout errors within the physical-layer driver.

LC-7(2) poll_fnct should be called with p_phy_data as its sole argument. If it returns DEF_OK, then the device is ready and the function should return DEF_OK.

LC-7(3) If to_us microseconds elapse without the poll function or hardware ready/busy signaling indicating success, the function should return DEF_FAIL.

C-11 NOR Flash SPI BSP

The NOR driver must adapt to the specific hardware using a BSP. A serial NOR Flash will be interfaced on a SPI bus. See Appendix C, “SPI BSP” on page 494 for the details on how to implement the software port for your SPI bus.

C-12 SD/MMC Cardmode BSP

The SD/MMC cardmode protocol is unique to SD- and MMC-compliant devices. The generic driver handles the peculiarities for initializing, reading and writing a card (including state transitions and error handling), but each CPU has a different host controller that must be individually ported. To that end, a BSP, supplementary to the general µC/FS BSP, is required that abstracts the SD/MMC interface. The port includes one code file:

`FS_DEV_SD_CARD_BSP.C`

This file is generally placed with other BSP files in a directory named according to the following rubric:

```
\Micrium\Software\EvalBoards\<manufacturer>\<board_name>
\<compiler>\BSP\
```

Several example ports are included in the µC/FS distribution in files named according to the following rubric:

```
\Micrium\Software\uC-FS\Examples\BSP\Dev\SD\Card\<cpu_name>
```

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.

Function	Description
`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 C-7 **SD/MMC cardmode BSP functions**

Each BSP must implement the functions in Table C-7. (For information about creating a port for a platform accessing a SD/MMC device in SPI mode, see section C-13 "SD/MMC SPI mode BSP" on page 493) This software interface was designed by reviewing common host implementations as well as the SD card association's SD Specification Part A2 – SD Host Controller Simplified Specification, Version 2.00, which recommends a host architecture and provides the state machines that would guide operations. Example function implementations for a theoretical compliant host are provided in this chapter. Common advanced requirements (such as multiple cards per slot) and optimizations (such as DMA) are possible. No attempt has been made, however, to accommodate non-storage devices that are accessed on a SD/MMC cardmode, including SDIO devices.

The core operation being abstracted is the command/response sequence for high-level card transactions. The key functions, **`CmdStart()`**, **`CmdWaitEnd()`**, **`CmdDataRd()`** and **`CmdDataWr()`**, are called within the state machine of Figure C-5. If 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 handled if an operation cannot be completed.

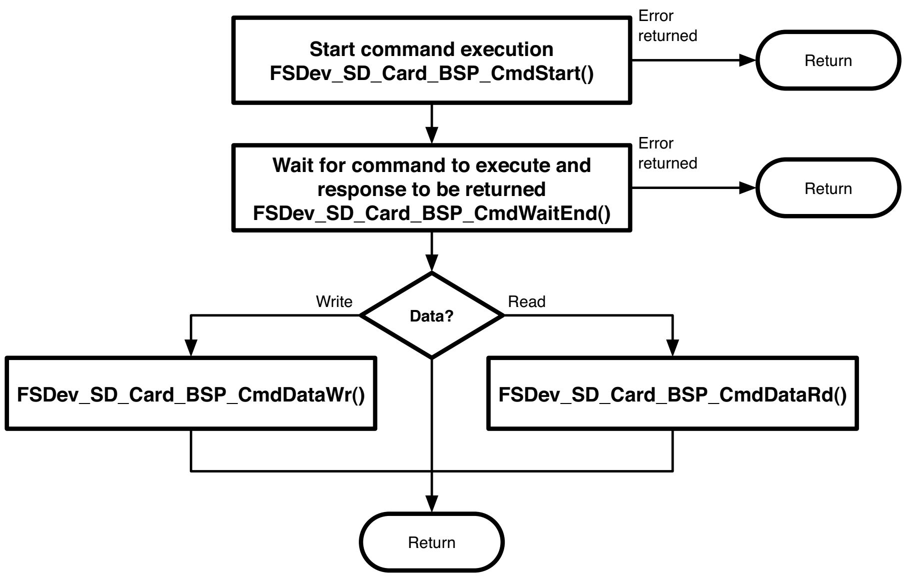


Figure C-5 **Command execution**

The remaining functions either investigate host capabilities (`GetBlkCntMax()`, `GetBusWidthMax()`) or set operational parameters (`SetBusWidth()`, `SetClkFreq()`, `SetTimeoutData()`, `SetTimeoutResp()`). Together, these function sets help configure a new card upon insertion. Note that the parameters configured by the 'set' functions belong to the card, not the slot; if multiple cards may be multiplexed in a single slot, these must be saved when set and restored whenever `Lock()` is called.

Two elements of host behavior routinely influence implementation and require design choices. First, block data can typically be read/written either directly from a FIFO or transferred automatically by the peripheral to/from a memory buffer with DMA. While the former approach may be simpler—no DMA controller need be setup—it may not be reliable. Unless the host can stop the host clock upon FIFO underrun (for write) or overrun (for read), effectively pausing the operation from the card's perspective, transfers at high clock frequency or multiple-bus configurations will probably fail. Interrupts or other tasks can interrupt the operation, or the CPU just may be unable to fill the FIFO fast enough. DMA avoids those pitfalls by offloading the responsibility for moving data directly to the CPU.

Second, the completion of operations such as command execution and data read/write are often signaled via interrupts (unless some error occurs, whereupon a different interrupt is triggered). During large transfers, these operations occur frequently and the typical wait between initiation and completion is measured in microseconds. On most platforms, polling the interrupt status register within the task performs better (i.e., results in faster reads and writes) than waiting on a semaphore for an asynchronous notification from the ISR, because the penalty of extra context switches is not incurred.

C-12-1 FSDev_SD_Card_BSP_Open()

```
CPU_BOOLEAN  FSDev_SD_Card_BSP_Open (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_sd_card_bsp.c`	`FSDev_SD_Card_Refresh()`	N/A

Open (initialize) SD/MMC card interface.

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.

C-12-2 FSDev_SD_Card_BSP_Lock()

```
FSDev_SD_Card_BSP_Unlock()
void  FSDev_SD_Card_BSP_Lock   (FS_QTY  unit_nbr);
void  FSDev_SD_Card_BSP_Unlock (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_sd_card_bsp.c`	SD/MMC cardmode driver	N/A

Acquire/release SD/MMC card bus lock.

ARGUMENTS

`unit_nbr` Unit number of SD/MMC card.

RETURNED VALUE

None.

NOTES/WARNINGS

`FSDev_SD_Card_BSP_Lock()` will be called before the 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.

The clock frequency, bus width and timeouts set by the `FSDev_SD_Card_BSP_Set####()` functions are parameters of the card, not the bus. If multiple cards are located on the same bus, those parameters must be saved (in memory) when set and restored when `FSDev_SD_Card_BSP_Lock()` is called.

C-12-3 FSDev_SD_Card_BSP_CmdStart()

```
void  FSDev_SD_Card_BSP_CmdStart (FS_QTY              unit_nbr,
                                  FS_DEV_SD_CARD_CMD  *p_cmd,
                                  void                *p_data,
                                  FS_ERR              *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	SD/MMC cardmode driver	N/A

Start a command.

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 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_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 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.

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: Normal Response Command.
`FS_DEV_SD_CARD_RESP_TYPE_R1B`	R1b response.
`FS_DEV_SD_CARD_RESP_TYPE_R2`	R2 response: CID, CSD Register.
`FS_DEV_SD_CARD_RESP_TYPE_R3`	R3 response: OCR Register.
`FS_DEV_SD_CARD_RESP_TYPE_R4`	R4 response: Fast I/O Response (MMC).
`FS_DEV_SD_CARD_RESP_TYPE_R5`	R5 response: Interrupt Request Response (MMC).
`FS_DEV_SD_CARD_RESP_TYPE_R5B`	R5B response.
`FS_DEV_SD_CARD_RESP_TYPE_R6`	R6 response: Published RCA Response.
`FS_DEV_SD_CARD_RESP_TYPE_R7`	R7 response: Card Interface Condition.

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, `p_data`, is given so that a DMA transfer can be set up.

EXAMPLE

The example implementation of `FSDev_SD_Card_BSP_CmdStart()` in , like the examples in subsequent sections, targets a generic host conformant to the SD card association's host controller specification. While few hosts do conform, most have a similar mixture of registers and registers fields and require the same sequences of basic actions.

```
void  FSDev_SD_Card_BSP_CmdStart (FS_QTY               unit_nbr,
                                  FS_DEV_SD_CARD_CMD  *p_cmd,
                                  void                *p_data,
                                  FS_ERR              *p_err)
{
    CPU_INT16U  command;
    CPU_INT32U  present_state;
    CPU_INT16U  transfer_mode;
    present_state = REG_STATE;                     /* Chk if controller busy.    */  (1)
    if (DEF_BIT_IS_SET_ANY(present_state, BIT_STATE_CMD_INHIBIT_DAT |
                                          BIT_STATE_CMD_INHIBIT_CMD) == DEF_YES) {
       *p_err = FS_DEV_SD_CARD_ERR_BUSY;
        return;
    }
    transfer_mode = DEF_BIT_NONE;                 /* Calc transfer mode reg value. */  (2)
    if (p_cmd->DataType == FS_DEV_SD_CARD_DATA_TYPE_MULTIPLE_BLOCK) {
        transfer_mode |= BIT_TRANSFER_MODE_MULTIPLE_BLOCK
                      |  BIT_TRANSFER_MODE_AUTO_CMD12
                      |  BIT_TRANSFER_MODE_BLOCK_COUNT_ENABLE;
    }
    if (p_cmd->DataDir == FS_DEV_SD_CARD_DATA_DIR_CARD_TO_HOST) {
        transfer_mode |= BIT_TRANSFER_MODE_READ | BIT_TRANSFER_MODE_DMA_ENABLE;
    } else {
        transfer_mode |= BIT_TRANSFER_MODE_DMA_ENABLE;
    }
    command = (CPU_INT16U)p_cmd->Cmd << 8;        /* Calc command register value  */  (3)
    if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_DATA_START) == DEF_YES) {
        command |= BIT_COMMAND_DATA_PRESENT;
    }
    if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_IX_VALID) == DEF_YES) {
        command |= BIT_COMMAND_DATA_COMMAND_IX_CHECK;
    }
    if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_CRC_VALID) == DEF_YES) {
        command |= BIT_COMMAND_DATA_COMMAND_CRC_CHECK;
    }
    if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_RESP) == DEF_YES) {
        if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_RESP_LONG) == DEF_YES) {
            command |= BIT_COMMAND_DATA_COMMAND_RESPONSE_LENGTH_136;
        } else {
            if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_BUSY) == DEF_YES) {
                command |= BIT_COMMAND_DATA_COMMAND_RESPONSE_LENGTH_48;
            } else {
                command |= BIT_COMMAND_DATA_COMMAND_RESPONSE_LENGTH_48_BUSY;
            }
        }
    }
```

```
                                                  /* Write registers to exec cmd. */  (4)
    REG_SDMA_ADDESS   = p_data;
    REG_BLOCK_COUNT   = p_cmd->BlkCnt;
    REG_BLOCK_SIZE    = p_cmd->BlkSize;
    REG_ARGUMENT      = p_cmd->Arg;
    REG_TRANSFER_MODE = transfer_mode;
    REG_COMMAND       = command;
   *p_err = FS_DEV_SD_CARD_ERR_NONE;
}
```

Listing C-8 **FSDev_SD_Card_BSP_CmdStart()**

LC-8(1) Check whether the controller is busy. Though no successful operation should return without the controller idle, an error condition, programming mistake or unexpected condition could make an assumption about initial controller state false. This simple validation is recommended to avoid side-effects and to aid port debugging.

LC-8(2) Calculate the transfer mode register value. The command's DataType and DataDir members specify the type and direction of any transfer. Since this examples uses DMA, DMA is enabled in the transfer mode register.

LC-8(3) Calculate the command register value. The command index is available in the command's Cmd member, which is supplemented by the bits OR'd into Flags to describe the expected result—response and data transfer—following the command execution.

LC-8(4) The hardware registers are written to execute the command. The sequence in which the registers are written is important. Typically, as in this example, the assignment to the command register actually triggers execution.

C-12-4 FSDev_SD_Card_BSP_CmdWaitEnd()

```
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);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	SD/MMC cardmode driver	N/A

Wait for command to end and get command response.

ARGUMENTS

unit_nbr Unit number of SD/MMC card.

p_cmd Pointer to command that is ending.

p_resp 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 error.

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 The data stored in the response buffer should include only the response data, i.e., should not include the start bit, transmission bit, command index, CRC and end bit.

a. For a command with a normal (48-bit) response, a 4-byte response should be stored in p_resp.

b. For a command with a long (136-bit) response, a 16-byte response should be returned 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 four 4-byte word should hold bits 31.. 0 of the response.

EXAMPLE

The implementation of `FSDev_SD_Card_BSP_CmdWaitEnd()` in is targeted for the same host controller as the other listings in this chapter; for more information, see `FSDev_SD_Card_BSP_CmdStart()`.

```
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)
{
    CPU_INT16U  interrupt_status;
    CPU_INT16U  error_status;
    CPU_INT16U  timeout;
    timeout          = 0u;                        /* Wait until cmd exec complete.*/  (1)
    interrupt_status = REG_INTERRUPT_STATUS;
    while (DEF_BIT_IS_CLR(interrupt_status, BIT_INTERRUPT_STATUS_ERROR |
                                       BIT_INTERRUPT_STATUS_COMMAND_COMPLETE) == DEF_YES)) {
        timeout++;
        interrupt_status = REG_INTERRUPT_STATUS;
        if (timeout == TIMEOUT_RESP_MAX) {
           *p_err = FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT;
            return;
        }
    }
                                                  /* Handle error.              */  (2)
    if (DEF_BIT_IS_SET(interrupt_status, BIT_INTERRUPT_STATUS_ERROR) == DEF_YES) {
        error_status = REG_ERROR_STATUS;
        if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_COMMAND_INDEX) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_RESP_CMD_IX;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_COMMAND_END_BIT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_RESP_END_BIT;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_COMMAND_CRC) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_RESP_CRC;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_COMMAND_TIMEOUT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_RESP_TIMEOUT;
        } else {
           *p_err = FS_DEV_SD_CARD_ERR_RESP;
        }
        REG_ERROR_STATUS     = error_status;
        REG_INTERRUPT_STATUS = interrupt_status;
        return;
    }
```

```
                                                /* Read response.            */  (3)
    REG_INTERRUPT_STATUS = BIT_INTERRUPT_STATUS_COMMAND_COMPLETE;
    if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_RESP) == DEF_YES) {
        if (DEF_BIT_IS_SET(p_cmd->Flags, FS_DEV_SD_CARD_CMD_FLAG_RESP_LONG) == DEF_YES) {
           *(p_resp + 3) = REG_RESPONSE_00
           *(p_resp + 2) = REG_RESPONSE_01
           *(p_resp + 1) = REG_RESPONSE_02
           *(p_resp + 0) = REG_RESPONSE_03
        } else {
           *(p_resp + 0) = REG_RESPONSE_00
        }
    }

   *p_err = FS_DEV_SD_CARD_ERR_NONE;
}
```

Listing C-9 **FSDev_SD_Card_BSP_CmdWaitEnd()**

LC-9(1) Wait until command execution completes or an error occurs. The wait loop (or wait on semaphore) SHOULD always have a timeout to avoid blocking the task in the case of an unforeseen hardware malfunction or a software flaw.

LC-9(2) Check if an error occurred. The error status register is decoded to produce the actual error condition. That is not necessary, strictly, but error counters that accumulate within the generic driver based upon returned error values may be useful while debugging a port.

LC-9(3) Read the response, if any. Note that the order in which a long response is stored in the buffer may oppose its storage in the controller's register or FIFO.

C-12-5 FSDev_SD_Card_BSP_CmdDataRd()

```
void  FSDev_SD_Card_BSP_CmdDataRd (FS_QTY               unit_nbr,
                                   FS_DEV_SD_CARD_CMD  *p_cmd,
                                   void                *p_dest,
                                   FS_ERR              *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_RdData()	N/A

Read data following a command.

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 error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

EXAMPLE

The implementation of **FSDev_SD_Card_BSP_CmdDataRd()** in Listing C-10 is targeted for the same host controller as the other listings in this chapter; for more information, see **FSDev_SD_Card_BSP_CmdStart()**.

```
void  FSDev_SD_Card_BSP_CmdDataRd (FS_QTY              unit_nbr,
                                   FS_DEV_SD_CARD_CMD  *p_cmd,
                                   void                *p_dest,
                                   FS_ERR              *p_err)
{
    CPU_INT16U  interrupt_status;
    CPU_INT16U  error_status;
    CPU_INT16U  timeout;
    timeout          = 0u;                        /* Wait until data xfer compl. */  (1)
    interrupt_status = REG_INTERRUPT_STATUS;
    while (DEF_BIT_IS_CLR(interrupt_status,BIT_INTERRUPT_STATUS_ERROR |
                                          BIT_INTERRUPT_STATUS_TRANSFER_COMPLETE) == DEF_YES)) {
        timeout++;
        interrupt_status = REG_INTERRUPT_STATUS;
        if (timeout == TIMEOUT_TRANSFER_MAX) {
           *p_err = FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT;
            return;
        }
    }
                                                  /* Handle error.              */  (2)
    if (DEF_BIT_IS_SET(interrupt_status, BIT_INTERRUPT_STATUS_ERROR) == DEF_YES) {
        error_status = REG_ERROR_STATUS;
        if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_END_BIT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_CRC) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA_CRC;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_TIMEOUT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA_TIMEOUT;
        } else {
           *p_err = FS_DEV_SD_CARD_ERR_UNKONWN;
        }
        REG_ERROR_STATUS     = error_status;
        REG_INTERRUPT_STATUS = interrupt_status;
        return;
    }

   *p_err = FS_DEV_SD_CARD_ERR_NONE;                                                  (3)
}
```

Listing C-10 **FSDev_SD_Card_BSP_CmdDataRd()**

LC-10(1) Wait until data transfer completes or an error occurs. The wait loop (or wait on semaphore) SHOULD always have a timeout to avoid blocking the task in the case of an unforeseen hardware malfunction or a software flaw.

LC-10(2) Check if an error occurred. The error status register is decoded to produce the actual error condition. That is not necessary, strictly, but error counters that accumulate within the generic driver based upon returned error values may be useful while debugging a port.

LC-10(3) Return no error. The data has been transferred already to the memory buffer using DMA.

C-12-6 FSDev_SD_Card_BSP_CmdDataWr()

```
void  FSDev_SD_Card_BSP_CmdDataWr (FS_QTY              unit_nbr,
                                   FS_DEV_SD_CARD_CMD  *p_cmd,
                                   void                *p_src,
                                   FS_ERR              *p_err);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_WrData()	N/A

Write data following a command.

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 error.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

EXAMPLE

The implementation of **FSDev_SD_Card_BSP_CmdDataWr()** in Listing C-11 is targeted for the same host controller as the other listings in this chapter; for more information, see **FSDev_SD_Card_BSP_CmdStart()**.

```
void  FSDev_SD_Card_BSP_CmdDataWr (FS_QTY              unit_nbr,
                                   FS_DEV_SD_CARD_CMD  *p_cmd,
                                   void                *p_src,
                                   FS_ERR              *p_err)
{
    CPU_INT16U  interrupt_status;
    CPU_INT16U  error_status;
    CPU_INT16U  timeout;
    timeout          = 0u;                       /* Wait until data xfer compl. */  (1)
    interrupt_status = REG_INTERRUPT_STATUS;
    while (DEF_BIT_IS_CLR(interrupt_status,BIT_INTERRUPT_STATUS_ERROR |
                                        BIT_INTERRUPT_STATUS_TRANSFER_COMPLETE) == DEF_YES)) {
        timeout++;
        interrupt_status = REG_INTERRUPT_STATUS;
        if (timeout == TIMEOUT_TRANSFER_MAX) {
           *p_err = FS_DEV_SD_CARD_ERR_WAIT_TIMEOUT;
            return;
        }
    }
                                                 /* Handle error.              */  (2)
    if (DEF_BIT_IS_SET(interrupt_status, BIT_INTERRUPT_STATUS_ERROR) == DEF_YES) {
        error_status = REG_ERROR_STATUS;
        if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_END_BIT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_CRC) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA_CRC;
        } else if (DEF_BIT_IS_SET(error_status, REG_ERROR_STATUS_DATA_TIMEOUT) == DEF_YES) {
           *p_err = FS_DEV_SD_CARD_ERR_DATA_TIMEOUT;
        } else {
           *p_err = FS_DEV_SD_CARD_ERR_UNKONWN;
        }
        REG_ERROR_STATUS     = error_status;
        REG_INTERRUPT_STATUS = interrupt_status;
        return;
    }

   *p_err = FS_DEV_SD_CARD_ERR_NONE;                                                  (3)
}
```

Listing C-11 **FSDev_SD_Card_BSP_CmdDataWr()**

LC-11(1) Wait until data transfer completes or an error occurs. The wait loop (or wait on semaphore) SHOULD always have a timeout to avoid blocking the task in the case of an unforeseen hardware malfunction or a software flaw.

LC-11(2) Check if an error occurred. The error status register is decoded to produce the actual error condition. That is not necessary, strictly, but error counters that accumulate within the generic driver based upon returned error values may be useful while debugging a port.

LC-11(3) Return no error. The data has been transferred already from the memory buffer using DMA.

C-12-7 FSDev_SD_Card_BSP_GetBlkCntMax()

```
CPU_INT32U  FSDev_SD_Card_BSP_GetBlkCntMax (FS_QTY      unit_nbr,
                                            CPU_INT32U  blk_size);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_Refresh()	N/A

Get maximum number of blocks that can be transferred with a multiple read or multiple write command.

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 maximum 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 and re-opened for any changes to be effective.

C-12-8 FSDev_SD_Card_BSP_GetBusWidthMax()

```
CPU_INT08U  FSDev_SD_Card_BSP_GetBusWidthMax (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_sd_card_bsp.c`	`FSDev_SD_Card_Refresh()`	N/A

Get maximum bus width, in bits.

ARGUMENTS

`unit_nbr` Unit number of SD/MMC card.

RETURNED VALUE

Maximum bus width.

NOTES/WARNINGS

1 Legal values are typically 1, 4 and 8.

2 This function SHOULD always return the same value. If hardware constraints change at run-time, the device MUST be closed and re-opened for any changes to be effective.

C-12-9 FSDev_SD_Card_BSP_SetBusWidth()

```
void  FSDev_SD_Card_BSP_SetBusWidth (FS_QTY      unit_nbr,
                                     CPU_INT08U  width);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_Refresh(), FSDev_SD_Card_SetBusWidth()	N/A

Set bus width.

ARGUMENTS

`unit_nbr` Unit number of SD/MMC card.

`width` Bus width, in bits.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

EXAMPLE

The implementation of **FSDev_SD_Card_BSP_SetBusWidth()** in Listing C-12 is targeted for the same host controller as the other listings in this chapter; for more information, see **FSDev_SD_Card_BSP_CmdStart()**.

```
void  FSDev_SD_Card_BSP_SetBusWidth (FS_QTY      unit_nbr,
                                     CPU_INT08U  width)
{
    if (width == 1u) {
        REG_HOST_CONTROL &= ~BIT_HOST_CONTROL_DATA_TRANSFER_WIDTH;
    } else {
        REG_HOST_CONTROL |=  BIT_HOST_CONTROL_DATA_TRANSFER_WIDTH;
    }
}
```

Listing C-12 **FSDev_SD_Card_BSP_SetBusWidth()**

C-12-10 FSDev_SD_Card_BSP_SetClkFreq()

```
void  FSDev_SD_Card_BSP_SetClkFreq (FS_QTY      unit_nbr,
                                    CPU_INT32U  freq);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_Refresh()	N/A

Set clock frequency.

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.

C-12-11 FSDev_SD_Card_BSP_SetTimeoutData()

```
void  FSDev_SD_Card_BSP_SetTimeoutData (FS_QTY      unit_nbr,
                                        CPU_INT32U  to_clks);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_Refresh()	N/A

Set data timeout.

ARGUMENTS

`unit_nbr` Unit number of SD/MMC card.

`to_clks` Timeout, in clocks.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-12-12 FSDev_SD_Card_BSP_SetTimeoutResp()

```
void  FSDev_SD_Card_BSP_SetTimeoutResp (FS_QTY      unit_nbr,
                                        CPU_INT32U  to_ms);
```

File	Called from	Code enabled by
fs_dev_sd_card_bsp.c	FSDev_SD_Card_Refresh()	N/A

Set data timeout.

ARGUMENTS

`unit_nbr` Unit number of SD/MMC card.

`to_ms` Timeout, in milliseconds.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-13 SD/MMC SPI mode BSP

SD/MMC card can also be accessed through an SPI bus (also described as the one-wire mode). Please refer to section C-14 "SPI BSP" on page 494 for the details on how to implement the software port for your SPI bus.

C-14 SPI BSP

Among the most common—and simplest—serial interfaces supported by built-in CPU peripherals is Serial Peripheral Interface (SPI). Four hardware signals connect a defined master (or host) to each slave (or device): a slave select, a clock, a slave input and a slave output. Three of these, all except the slave select, may be shared among all slaves, though hosts often have several SPI controllers to simplify integration and allow simultaneous access to multiple slaves. Serial flash, serial EEPROM and SD/MMC cards are among the many devices which use SPI.

Signal	Description
`SSEL (CS)`	Slave select
`SCLK`	Clock
`SO (MISO)`	Slave output (master input)
`SI (MOSI)`	Slave input (master output)

Table C-8 **SPI signals**

No specification exists for SPI, a condition which invites technological divergence. So though the simplicity of the interface limits variations between implementations, the required transfer unit length, shift direction, clock frequency and clock polarity and phase do vary from device to device. Take as an example Figure C-6 which gives the bit form of a basic command/response exchange on a typical serial flash. The command and response both divide into 8-bit chunks, the transfer unit for the device. Within these units, the data is transferred from most significant bit (MSB) to least significant bit (LSB), which is the slave's shift direction. Though not evident from the diagram—the horizontal axis being labeled in clocks rather than time—the slave cannot operate at a frequency higher than 20-MHz. Finally, the clock signal prior to slave select activation is low (clock polarity or CPOL is 0), and data is latched on the rising clock edge (clock phase or CPHA is 0). Together, those are the aspects of SPI communication that may need to be configured:

- Transfer unit length. A transfer unit is the underlying unit of commands, responses and data. The most common value is eight bits, though slaves commonly require (and masters commonly support) between 8 and 16 bits.

- Shift direction. Either the MSB or LSB of each transfer unit can be the first transmitted on the data line.

- Clock frequency. Limits are usually imposed upon the frequency of the clock signal. Of all variable SPI communication parameters, only this one is explicitly set by the device driver.

- 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:

  - If CPOL = 0, the clock is low when inactive.

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

  - If CPHA = 0, data is “read” on the leading edge of the clock and “changed” on the following edge.

    If CPHA = 1, data is “changed” on the leading edge of the clock and “read” on the leading edge.

  The most commonly-supported settings are {CPOL, CPHA} = {0, 0} and {1, 1}.

- Slave select polarity. The “active” level of the slave select may be electrically high or low. Low is ubiquitous, high rare.

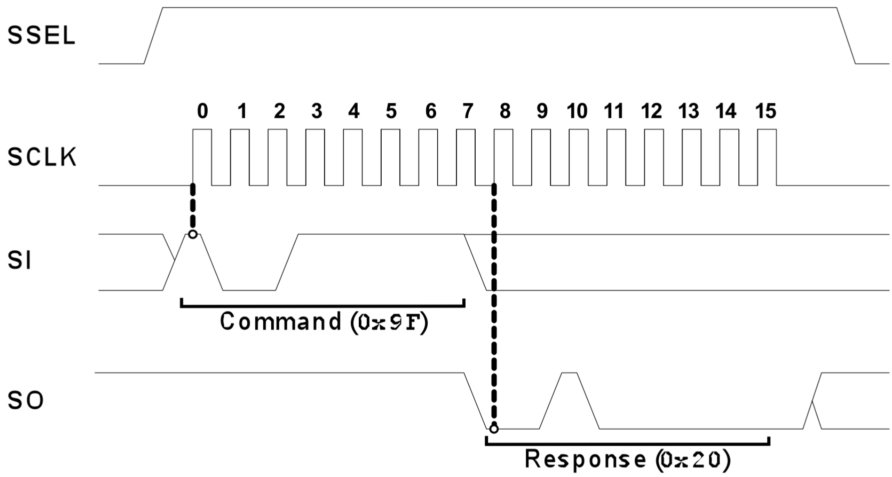


Figure C-6 **Example SPI transaction**

A BSP is required that abstracts a CPU's SPI peripheral. The port includes one code file named according to the following rubric:

`FS_DEV_<dev_name>_BSP.C` or `FS_DEV_<dev_name>_SPI_BSP.c`

This file is generally placed with other BSP files in a directory named according to the following rubric:

```
\Micrium\Software\EvalBoards\<manufacturer>\<board_name>
\<compiler>\BSP\
```

Several example ports are included in the µC/FS distribution in files named according to the following rubric:

```
\Micrium\Software\uC-FS\Examples\BSP\Dev\NAND\<manufacturer>\<cpu_name>
\Micrium\Software\uC-FS\Examples\BSP\Dev\NOR\<manufacturer>\<cpu_name>
\Micrium\Software\uC-FS\Examples\BSP\Dev\SD\SPI\<manufacturer>\<cpu_name>
```

Check all of these directories for ports for a CPU if porting any SPI device; the CPU may be been used with a different type of device, but the port should support another with none or few modifications. Each port must implement the functions to be placed into a `FS_DEV_SPI_API` structure:

```
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
};
```

The functions which must be implemented are listed and described in Table C-9. SPI is no more than a physical interconnect. The protocol of command-response interchange the master follows to control a slave is specified on a per-slave basis. Control of the chip select (SSEL) is separated from the reading and writing of data to the slave because multiple bus transactions (e.g., a read then a write then another read) are often performed without breaking slave selection. Indeed, some slaves require bus transactions (or "empty" clocks) AFTER the select has been disabled.

Function	**Description**
`Open()`	Open (initialize) hardware for SPI.
`Close()`	Close (uninitialize) hardware for SPI.
`Lock()`	Acquire SPI bus lock.
`Unlock()`	Release SPI bus lock.
`Rd()`	Read from SPI bus.
`Wr()`	Write to SPI bus.
`ChipSelEn()`	Enable device chip select.
`ChipSelDis()`	Disable device chip select
`SetClkFreq()`	Set SPI clock frequency

Table C-9 **SPI port functions**

The first argument of each of these port functions is the device unit number, an identifier unique to each driver/device type—after all, it is the number in the device name. For example, “sd:0:” and “nor:0:” both have unit number 1. If two SPI devices are located on the same SPI bus, either of two approaches can resolve unit number conflicts:

- Unique unit numbers. All devices on the same bus can use the same SPI BSP if and only if each device has a unique unit number. For example, the SD/MMC card “sd:0:” and serial NOR “nor:1:” require only one BSP.

- Unique SPI BSPs. Devices of different types (e.g., a SD/MMC card and a serial NOR) can have the same unit number if and only if each device uses a separate BSP. For example, the SD/MMC card “sd:0:” and serial “nor:0:” require separate BSPs.

C-14-1 Open()

```
CPU_BOOLEAN  FSDev_BSP_SPI_Open (FS_QTY unit_nbr);
```

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Open (initialize) hardware for SPI.

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, including:

a. Transfer unit length

b. Shift direction

c. Clock frequency

d. Clock polarity and phase (CPOL and CPHA)

e. Slave select polarity

3 For a SD/MMC card, the following settings should be used:

a. Transfer unit length: 8-bits

b. Shift direction: MSB first

c. Clock frequency: 400-kHz (initially)

d. Clock polarity and phase (CPOL and CPHA): CPOL = 0, CPHA = 0

e. Slave select polarity: active low.

4 The slave select (SSEL or CS) MUST be configured as a GPIO output; it should not be controlled by the CPU's SPI peripheral. The SPI port's **`ChipSelEn()`** and **`ChipSelDis()`** functions manually enable and disable the SSEL.

C-14-2 Close()

```
void  FSDev_BSP_SPI_Close (FS_QTY unit_nbr);
```

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Close (uninitialize) hardware for SPI.

ARGUMENTS

`unit_nbr` Unit number of device.

RETURNED VALUE

None.

NOTES/WARNINGS

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

C-14-3 Lock() / Unlock()

```
void  FSDev_BSP_SPI_Lock   (FS_QTY unit_nbr);
void  FSDev_BSP_SPI_Unlock (FS_QTY unit_nbr);
```

File	Called from	Code enabled by
`fs_dev_<dev_name>_bsp.c`	Device driver	N/A

Acquire/release SPI bus lock.

ARGUMENTS

`unit_nbr` Unit number of device.

RETURNED VALUE

None.

NOTES/WARNINGS

`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 `Unlock()` has been made.

The clock frequency set by the `SetClkFreq()` function is a parameter of the device, not the bus. If multiple devices are located on the same bus, those parameters must be saved (in memory) when set and restored by `Lock()`. The same should be done for initialization parameters such as transfer unit size and shift direction that vary from device to device.

C-14-4 Rd()

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

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Read from SPI bus.

ARGUMENTS

`unit_nbr` Unit number of device.

`p_dest` Pointer to destination buffer.

`cnt` Number of octets to read.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-14-5 Wr()

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

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Write to SPI bus.

ARGUMENTS

unit_nbr Unit number of device.

p_src Pointer to source buffer.

cnt Number of octets to write.

RETURNED VALUE

None.

NOTES/WARNINGS

None.

C-14-6 ChipSelEn() /ChipSelDis()

```
void  FSDev_BSP_SPI_ChipSelEn  (FS_QTY  unit_nbr);
void  FSDev_BSP_SPI_ChipSelDis (FS_QTY  unit_nbr);
```

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Enable/disable device chip select.

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; to disable the device, the chip select pin should be set.

C-14-7 SetClkFreq()

```
void  FSDev_BSP_SPI_SetClkFreq (FS_QTY      unit_nbr,
                                CPU_INT32U  freq);
```

File	Called from	Code enabled by
fs_dev_<dev_name>_bsp.c	Device driver	N/A

Set SPI clock frequency.

ARGUMENTS

`unit_nbr` Unit number of device.

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.

Appendix

D

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.

D-1 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.

D-2 FS_DEV_INFO

```
typedef struct fs_dev_info {
    FS_STATE     State;
    FS_SEC_QTY   Size;
    FS_SEC_SIZE  SecSize;
    CPU_BOOLEAN  Fixed;
  } 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_CLOSED`	Device is closed.
`FS_DEV_STATE_CLOSING`	Device is closing.
`FS_DEV_STATE_OPENING`	Device is opening.
`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 low-level format is valid.

`Size` The number of sectors on the device.

`SecSize` The size of each device sector.

`Fixed` Indicates whether the device is fixed or removable.

NOTES

None.

D-3 FS_DEV_NAND_CFG

```
typedef struct fs_dev_nand_cfg {
    CPU_INT32U           BlkNbrFirst;
    FS_SEC_SIZE          SecSize;
    CPU_INT32U           BlkCnt;
    CPU_INT08U           RBCnt;
    FS_DEV_NAND_PHY_API *PhyPtr;
    CPU_INT08U           BusWidth;
    CPU_INT32U           MaxClkFreq;
  } FS_DEV_NAND_CFG;
```

File	Used for
fs_dev_nand.h	Second argument of `FSDev_Open()` (when opening a NAND device)

Configures the properties of a NAND device that will be opened. A pointer to this structure is passed as the second argument of `FSDev_Open()` for a NAND device.

MEMBERS

`BlkNbrFirst` MUST specify which block of the NAND flash memory will be the first used for the file system data.

`SecSize` MUST specify the sector size in bytes for the NAND flash (either 512, 1024, 2048 or 4096).

`BlkCnt` MUST specify the size of the NAND flash in number of blocks.

`RBCnt` MUST specify the number of replacement blocks that will be used by the driver.

`PhyPtr` MUST point to the appropriate physical-layer driver:

`FSDev_NAND_0512x08`	512-byte page NAND, 8-bit data bus.
`FSDev_NAND_2048x08`	2048-byte page NAND, 8-bit data bus.
`FSDev_NAND_2048x16`	2048-byte page NAND, 16-bit data bus.
`FSDev_NAND_AT45`	Atmel AT45 serial DataFlash

	Other	User-developed

`BusWidth` is the bus width, in bits, between the MCU/MPU and each connected device.

MaxClkFreq For a serial flash, the maximum clock frequency is specified via **`MaxClkFreq`**.

NOTES

None.

D-4 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.

D-5 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.

D-6 FS_DIR_ENTRY (struct fs_dirent)

```
typedef struct fs_dirent {
    CPU_CHAR       Name[FS_CFG_MAX_FILE_NAME_LEN + 1u];
    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 D-2 "FS_DEV_INFO" on page 510

NOTES

None.

D-7 FS_ENTRY_INFO

```
typedef struct fs_entry_info {
    FS_FLAGS      Attrib;
    FS_FILE_SIZE  Size;
    CLK_TS_SEC    DateTimeCreate;
    CLK_TS_SEC    DateAccess;
    CLK_TS_SEC    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 7-2-1 "File and Directory Attributes" on page 104).

`Size` The size of the file, in octets.

`DateTimeCreate` The creation timestamp of the file or directory.

`DateAccess` The last access date 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.

D-8 FS_FAT_SYS_CFG

```
typedef struct fs_fat_sys_cfg {
    FS_SEC_QTY      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 9, “File Systems: FAT” on page 109.

D-9 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.

D-10 FS_VOL_INFO

```
typedef struct fs_vol_info {
    FS_STATE      State;
    FS_STATE      DevState;
    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_CLOSED`	Volume is closed.
`FS_VOL_STATE_CLOSING`	Volume is closing.
`FS_VOL_STATE_OPENING`	Volume is opening.
`FS_VOL_STATE_OPEN`	Volume is open.
`FS_VOL_STATE_PRESENT`	Volume device is present.
`FS_VOL_STATE_MOUNTED`	Volume is mounted.

`DevState` The device state:

`FS_DEV_STATE_CLOSED`	Device is closed.
`FS_DEV_STATE_CLOSING`	Device is closing.
`FS_DEV_STATE_OPENING`	Device is opening.
`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 low-level format is valid.

`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

E

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.

E-1 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 E-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 E-2 **Directory function exclusion**
These functions are NOT included if FS_CFG_DIR_EN is DEF_DISABLED

E-2 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 E-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 E-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 E-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 E-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_CONCURRENT_ENTRIES_ACCESS_EN

FS_CFG_CONCURRENT_ENTRIES_ACCESS_EN selects whether one file can be open multiple times (in one or more task). When enabled (DEF_ENABLED), files may be open concurrently mutliple times and without proection. When disabled (DEF_DISABLED), files may be open concurrently only in read-only mode, but may not be open concurrently in write mode. This option makes the filesystem safer when disabled.

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

Function	File
`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 E-7 **Read only function exclusion (continued)**
These functions are NOT included if FS_CFG_RD_ONLY_EN is DEF_ENABLED.

E-3 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.

E-4 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.

E-5 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.:

E-6 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.

E-7 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 E-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 E-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.

E-8 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.

E-9 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 E-9 **Trace Levels**

Appendix F

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.

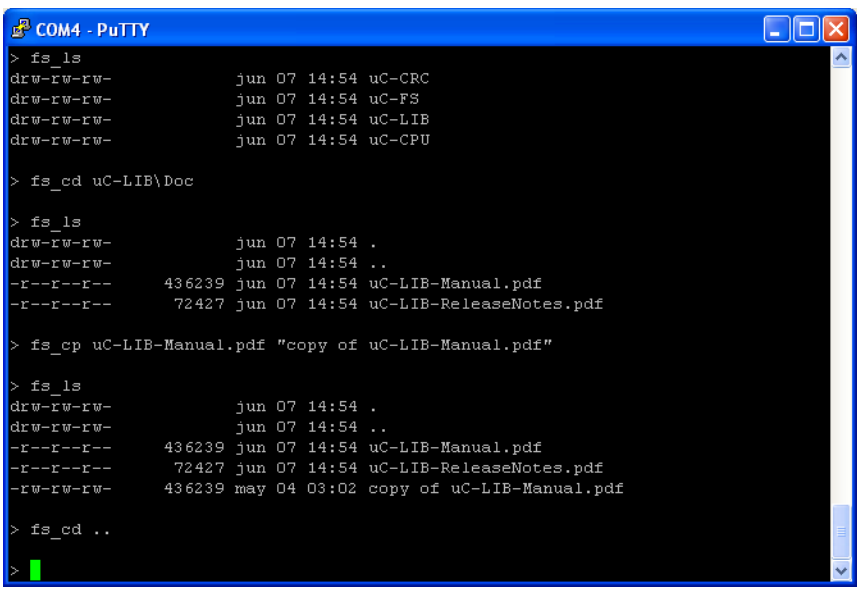


Figure F-1 **µC/FS shell command usage**

F-1 FILES AND DIRECTORIES

µC/FS with the shell commands (and µC/Shell) is organized into the directory structure shown in Figure F-2. The files constituting the shell commands ares outlined in this section; the generic file-system files, outlined in Chapter 3, “Directories and Files” on page 28, are also required.

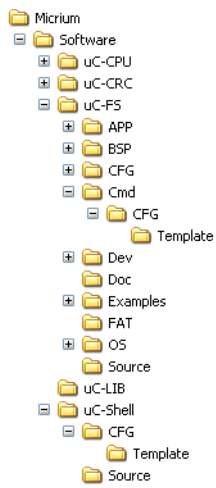


Figure F-2 **Directory Structure.**

`\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.

F-2 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`.
- fs_ `shell.h`.
- `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 F-1. The file system initialization (`FS_Init()`) function should have previously been called.

```
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);
}
```

Listing F-1 **Initializing µC/Shell**

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 F-2.

```
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. */
}
```

Listing F-2 **Executing shell commands & handling shell output.**

LF-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, "\".

LF-2(2) The next command, ending with a newline, should be read from the command line.

LF-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()**.

F-3 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 F-1 **Commands**

Information about each command can be obtained using the help (-h) option:

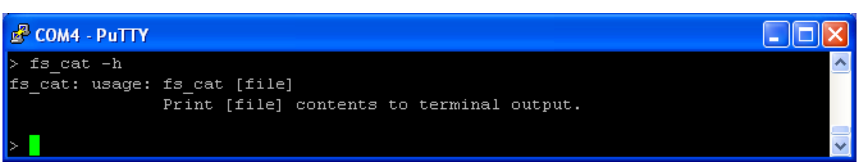


Figure F-3 **Help option output**

F-3-1 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.

F-3-2 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.

F-3-3 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`.

F-3-4 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 F-4 **fs_date output**

F-3-5 fs_df

Report disk free space.

USAGES

`fs_df`

`fs_df [vol]`

ARGUMENTS

volIf 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.

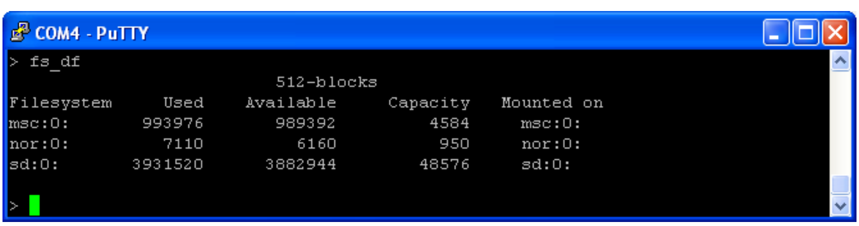


Figure F-5 **fs_df Output**

F-3-6 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.

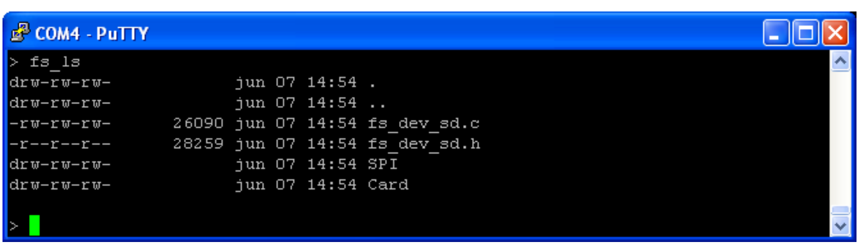


Figure F-6 **fs_ls Output**

F-3-7 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.

F-3-8 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.

F-3-9 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.

F-3-10 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.

F-3-11 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.

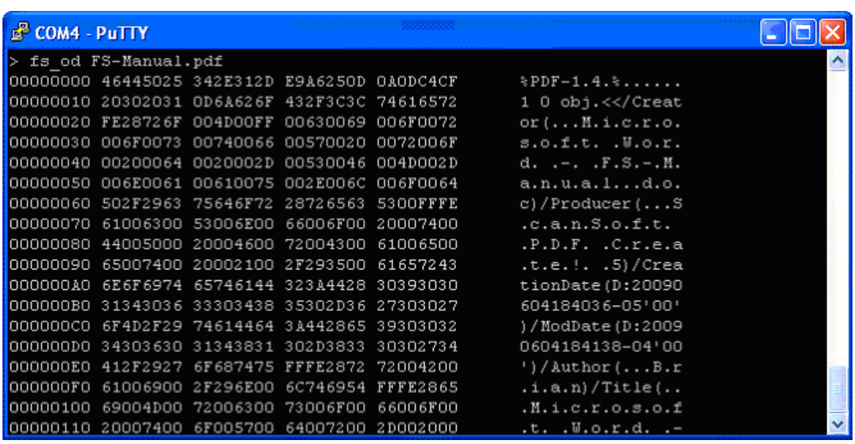


Figure F-7 **fs_od Output**

F-3-12 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.

F-3-13 fs_rm

Remove a file.

USAGES

`fs_rm [file]`

ARGUMENTS

`file` File 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.

F-3-14 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.

F-3-15 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.

F-3-16 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.

F-3-17 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.

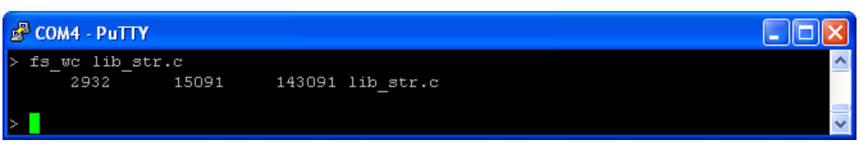


Figure F-8 **fs_wc Output**

F-4 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

G

Bibliography

Labrosse, Jean J. 2009, *µC/OS-III, The Real-Time Kernel*, Micrium Press, 2009, ISBN 978-0-98223375-3-0.

Légaré, Christian 2010, *µC/TCP-IP, The Embedded Protocol Stack*, Micrium Press, 2010, ISBN 978-0-98223375-0-9.

POSIX:2008 The Open Group Base Specifications Issue 7, IEEE Standard 1003.1-2008.

Programming Lauguages -- C, ISO/IEC 9899:1999.

The Motor Industry Software Reliability Association, *MISRA-C:2004*, Guidelines for the Use of the C Language in Critical Systems, October 2004. www.misra-c.com.

Appendix

H

µC/FS Licensing Policy

H-1 µC/FS LICENSING

H-1-1 µC/FS SOURCE CODE

This book contains µC/FS precompiled in linkable object form, an evaluation board and tools (compiler/assembler/linker/debugger). Use µC/FS for free, as long as it is only used with the evaluation board that accompanies this book. You will need to purchase a license when using this code in a commercial product, where the intent is to make a profit. Users do not pay anything beyond the price of the book, evaluation board and tools, as long as they are used for educational purposes.

You will need to license µC/FS if you intend to use µC/FS in a commercial product where you intend to make a profit. You need to purchase this license when you make the decision to use µC/FS in a design, not when you are ready to go to production.

If you are unsure about whether you need to obtain a license for your application, please contact Micriµm and discuss your use with a sales representative.

Contact Micriµm

Micriµm
1290 Weston Road, Suite 306
Weston, FL 33326

+1 954 217 2036
+1 954 217 2037 (FAX)

E-Mail: sales@Micriµm.com
Website: www.Micriµm.com

H-1-2 µC/FS MAINTENANCE RENEWAL

Licensing µC/FS provides one year of limited technical support and maintenance and source code updates. Renew the maintenance agreement for continued support and source code updates.Contact sales@Micriµm.com for additional information.

H-1-3 µC/FS SOURCE CODE UPDATES

If you are under maintenance, you will be automatically emailed when source code updates become available. You can then download your available updates from the Micriµm FTP server. If you are no longer under maintenance, or forget your Micriµm FTP username or password, please contact sales@Micriµm.com.

H-1-4 µC/FS SUPPORT

Support is available for licensed customers. Please visit the customer support section in www.Micriµm.com. If you are not a current user, please register to create your account. A web form will be offered to you to submit your support question,

Licensed customers can also use the following contact:

Contact Micriµm

Micriµm
1290 Weston Road, Suite 306
Weston, FL 33326

+1 954 217 2036
+1 954 217 2037 (FAX)

AMEYA360
Components Supply Platform

Authorized Distribution Brand：

Website：

Welcome to visit www.ameya360.com

Contact Us：

Address：

401 Building No.5, JiuGe Business Center, Lane 2301, Yishan Rd
Minhang District, Shanghai , China

Sales：

Direct +86 (21) 6401-6692

Email amall@ameya360.com

QQ 800077892

Skype ameyasales1 ameyasales2

Customer Service：

Email service@ameya360.com

Partnership：

Tel +86 (21) 64016692-8333

Email mkt@ameya360.com