SCIOPTA

High Performance
Real-Time Operating Systems

FAT Filesystem

User's Guide and
Reference Manual
Support

EU Headquarters

SCIOPTA Systems GmbH
Hauptstrasse 293
79576 Weil am Rhein
Germany
Tel. +49 7621 940 919 0
Fax +49 7621 940 919 19
email: sales@sciopta.com
www.sciopta.com

Corporate Headquarters

SCIOPTA Systems AG
Fiechthagstrasse 19
4103 Bottmingen
Switzerland
Tel. +41 61 423 10 62
Fax +41 61 423 10 63
email: sales@sciopta.com
www.sciopta.com

Document No. S13243RL1

1 Table of Contents

2 SCIOPTA Real-Time Operating System....7
2.1 Introduction....7
2.2 CPU Family....8
2.3 About This Manual....9
3 FAT Filesystem....11
3.1 Introduction....11
3.2 Types of filesystem objects....13
3.3 Getting filesystem root object....14
3.4 Path formats and filesystem tree....15
3.5 Adding physical drives to filesystem tree....17
3.6 Removing physical drives from the filesystem tree....18
3.7 Working with drives and partitions....20
3.8 Working with files....23
3.9 Working with directories....26
3.10 Freeing filesystem objects....28
3.11 Accidental drive removal....29
3.12 Filesystem error hook....30
3.13 Configuration....31
4 SAFE FAT Filesystem....41
4.1 Introduction....41
4.2 Requirements....41
4.3 Enabling SAFE FAT....41
4.4 Creating SAFE FAT filesystem....41
4.5 Working with directories....41
4.6 Working with files....41
4.7 Limitations....42
4.8 Precautions....42
5 Messages....43
5.1 Introduction....43
5.2 Messages classification....44
5.3 SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY....46
5.4 SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY....47
5.5 SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY....49
5.6 SDD_DEV_READ / SDD_DEV_READ_REPLY....52
5.7 SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY....54
5.8 SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY....56

5.9 SDD_FILE_RESIZE64 / SDD_FILE_RESIZE64_REPLY....58
5.10 SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY....59
5.11 SDD_FILE_SEEK64 / SDD_FILE_SEEK64_REPLY....61
5.12 SDD_MAN_ADD / SDD_MAN_ADD_REPLY....63
5.13 SDD_MAN_GET / SDD_MAN_GET_REPLY....65
5.14 SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY....68
5.15 SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY....70
5.16 SDD_MAN_NOTIFY_ADD / SDD_MAN_NOTIFY_ADD_REPLY....73
5.17 SDD_MAN_NOTIFY_RM / SDD_MAN_NOTIFY_RM_REPLY....75
5.18 SDD_MAN_RM / SDD_MAN_RM_REPLY....77
5.19 SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY....79
5.20 SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY....81
5.21 SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY....83
5.22 SDD_OBJ_SIZE64_GET / SDD_OBJ_SIZE64_GET_REPLY....85
5.23 SDD_OBJ_TAG_GET / SDD_OBJ_TAG_GET_REPLY....87
5.24 SDD_OBJ_TAG_SET / SDD_OBJ_TAG_SET_REPLY....89
5.25 SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY....91
5.26 SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY....93
5.27 SFS_ASSIGN / SFS_ASSIGN_REPLY....95
5.28 SFS_FDISK / SFS_FDISK_REPLY....97
5.29 SFS_FEOF / SFS_FEOF_REPLY....99
5.30 SFS_FFLUSH / SFS_FFLUSH_REPLY....100
5.31 SFS_FFIRST / SFS_FFIRST_REPLY....101
5.32 SFS_FNEXT / SFS_FNEXT_REPLY....103
5.33 SFS_FGETS / SFS_FGETS_REPLY....105
5.34 SFS_FORMAT / SFS_FORMAT_REPLY....107
5.35 SFS_FPUTC / SFS_FPUTC_REPLY....109
5.36 SFS_FPUTS / SFS_FPUTS_REPLY....111
5.37 SFS_FTELL / SFS_FTELL_REPLY....113
5.38 SFS_FTELL64 / SFS_FTELL64_REPLY....114
5.39 SFS_GETSHORTNAME / SFS_GETSHORTNAME_REPLY....115
5.40 SFS_MOUNT / SFS_MOUNT_REPLY....117
5.41 SFS_MOVE / SFS_MOVE_REPLY....119
5.42 SFS_UMOUNT / SFS_UMOUNT_REPLY....121
5.43 SFS_UNASSIGN / SFS_UNASSIGN_REPLY....123

6 Application Programmer Interface....125
6.1 Introduction....125
6.2 sfs_assign....126

6.3 sfs_chdir....128
6.4 sfs_copy....131
6.5 sfs_create....133
6.6 sfs_delete....135
6.7 sfs_deleteRecursive....138
6.8 sfs_fclose....140
6.9 sfs_fdisk....142
6.10 sfs_feof....144
6.11 sfs_fflush....146
6.12 sfs_fgets....148
6.13 sfs_findFirst....150
6.14 sfs_findNext....152
6.15 sfs_format....154
6.16 sfs_fputc....156
6.17 sfs_fputs....158
6.18 sfs_free....160
6.19 sfs_fread....162
6.20 sfs_fresize....164
6.21 sfs_fresize64....166
6.22 sfs_fseek....168
6.23 sfs_fseek64....171
6.24 sfs_ftell....174
6.25 sfs_ftell64....176
6.26 sfs_fwrite....178
6.27 sfs_get....180
6.28 sfs_getcwd....183
6.29 sfs_getFirst....185
6.30 sfs_getNext....187
6.31 sfs_getProperty....189
6.32 sfs_getShortName....193
6.33 sfs_ioctl....195
6.34 sfs_mount....198
6.35 sfs_move....201
6.36 sfs_move2....203
6.37 sfs_objType....205
6.38 sfs_open....206
6.39 sfs_fopen....208
6.40 sfs_setProperty....210

6.41 sfs_size64Bad....214
6.42 sfs_size64Free....216
6.43 sfs_size64Total....218
6.44 sfs_size64Used....220
6.45 sfs_sizeBad....222
6.46 sfs_sizeFree....224
6.47 sfs_sizeTotal....226
6.48 sfs_sizeUsed....228
6.49 sfs_timeGet....230
6.50 sfs_timeSet....232
6.51 sfs_umount....234
6.52 sfs_unassign....236
6.53 sfs_waitAdd....239
6.54 sfs_waitRm....241
7 Manual versions....243
7.1 Manual version 1.9....243
7.2 Manual version 1.8....243
7.3 Manual version 1.7....243
7.4 Manual version 1.6....243
7.5 Manual version 1.5....244
7.6 Manual version 1.4....244
7.7 Manual version 1.0....244
8 Index....245

2 SCIOPTA Real-Time Operating System

2.1 Introduction

SCIOPTA is a high performance fully pre-emptive real-time operating system for hard real-time application available for many target platforms.

Available modules:

- Pre-emptive Multitasking Real-Time Kernel
- SCIOPTA Memory Management System - Support for MMU/MPU
- Board Support Packages
- IPS - Internet Protocols (TCP/IP) including IPS Applications (Web Server, TFTP, FTP, DNS, DHCP, Telnet and SMTP)
- FAT File System
- Flash File System, NOR and NAND
- Universal Serial Bus, USB Device
- Universal Serial Bus, USB Host
- DRUID - System Level Debugger including kernel awareness packages for source debuggers
- SCIOPTA PEG - Embedded GUI
- CONNECTOR - support for distributed multi-CPU systems
- SCAPI - SCIOPTA API for Windows or LINUX host
- SCSIM - SCIOPTA Simulator

SCIOPTA Real-Time Operating System contains design objects such as SCIOPTA modules, processes, messages and message pools. SCIOPTA is designed on a message based architecture allowing direct message passing between processes. Messages are mainly used for interprocess communication and synchronization. SCIOPTA messages are stored and maintained in memory pools. The memory pool manager is designed for high performance and memory fragmentation is avoided. Processes can be grouped in SCIOPTA modules, which allows you to design a very modular system. Modules can be static or created and killed during run-time as a whole. SCIOPTA modules can be used to encapsulate whole system blocks (such as a communication stack) and protect them from other modules in the system.

The SCIOPTA Real-Time Kernel has a very high performance. The SCIOPTA architecture is specifically designed to provide excellent real-time performance and small size. Internal data structures, memory management, interprocess communication and time management are highly optimized. SCIOPTA Real-Time kernels will also run on small single-chip devices without MMU.

2.2 CPU Family

SCIOPTA is delivered for a specific CPU Family such as: ARM®7/9, ARM®11, ARM® Cortex-M™, ARM® Cortex™-R, ARM® Cortex™-A, Renesas RX, Freescale™ PowerPC, apm PowerPC, Freescale™ ColdFire and Marvell Xscale.

Please consult the latest version of the SCIOPTA Price List for the complete list.

2.3 About This Manual

The **SCIOPTA** Real-time Operating System is a message based RTOS and is therefore very well suited for distributed multi-CPU systems.

The purpose of this **FAT Filesystem - User´s Guide and Reference Manual** is to give all needed information how to use the **SCIOPTA** FAT Filesystem.

Please see also the other **SCIOPTA** manuals, mainly the **SCIOPTA** - Kernel, User’s Guide and Reference Manual.

This manual includes only target processor independent information. All target processor related information can be found in the **SCIOPTA - Target Manual** which is different for each **SCIOPTA** supported processor family and includes:

- Installation information
- Getting started examples
- Description of the system configuration (SCONF tool)
- Information about the system building procedures
- Description of the board support packages (BSP)
- List of distributed files
- Release notes and version history

3 FAT Filesystem

3.1 Introduction

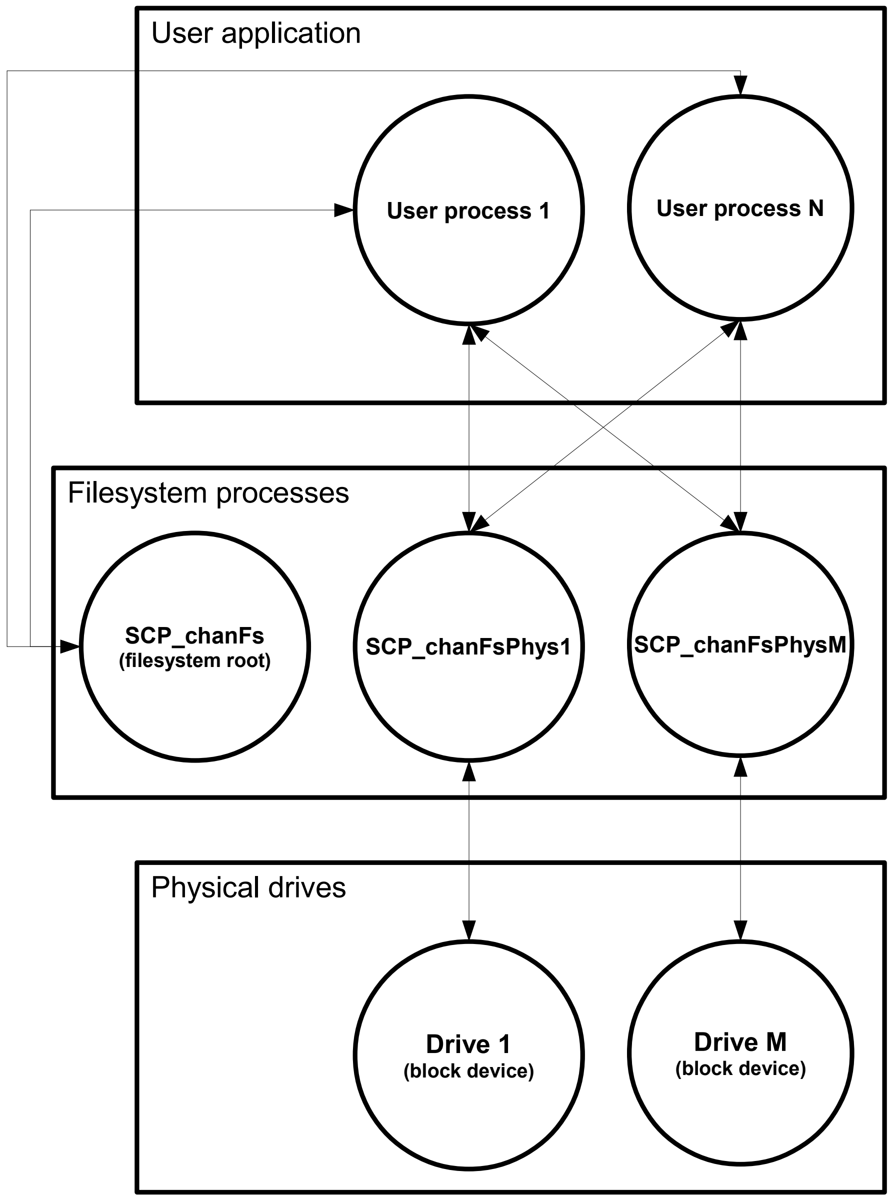
The filesystem described in this manual is a set of processes which together form an interface to FAT filesystems on physical drives. The interface is compatible with SCIOPTA device driver model. In addition, the interface defines filesystem specific messages for operations such as partitioning, formatting or mounting drives and partitions. The filesystem provides many configuration options, which allow user to adjust the filesystem for application requirements.

Following diagram shows the processes, which are part of the filesystem environment, and their relationship to the user processes and physical drives.

The **SCP_chanFs** process is the only one process which is persistent for all the use cases. Its main task is to create **SCP_chanFsPhys*** process for each physical drive added to the filesystem tree.

The **SCP_chanFsPhys*** processes are the interface to the FAT filesystems stored on physical drives. User applications can communicate with these processes by a set of filesystem functions (**sfs_*()**), or by using message interface directly. This documentation refers to theses processes as “**per-physical-drive-processes**”.

Physical drives processes are drivers such as an SD card driver, which expose the data in a block-oriented layout.


User application
User process 1
User process N
Filesystem processes
SCP_chanFs
(filesystem root)
SCP_chanFsPhys1
SCP_chanFsPhysM
Physical drives
Drive 1
(block device)
Drive M
(block device)

3.2 Types of filesystem objects

From the point of view of the user, the filesystem is a collection of objects of different types, which together form a tree-like structure. Following is a list of types of objects which are available in the filesystem tree:

- **SFS_ATTR_ROOT** – filesystem root. Refer to 3.3 for informations about filesystem root.
- **SFS_ATTR_DRIVE** – physical drive. Refer to 3.5 for informations about adding physical drives to the filesystem, to 3.6 for informations about removing physical drives from the filesystem and to 3.7 for informations about working with physical drives.
- **SFS_ATTR_PARTITION** – mounted FAT filesystem (entire physical drive or single partition). Refer to 3.7 for informations about working with partitions.
- **SFS_ATTR_DIR** – FAT directory. Refer to 3.9 for informations about working with directories.
- **SFS_ATTR_FILE** – FAT file or physical drive/partition accessed in raw mode. Refer to 3.8 for informations about working with files and to 3.7.12 for informations about raw access to physical drives and partitions.
- **SFS_ATTR_ANY** – means any type mentioned above. May be used in **sfs_get** function, which returns filesystem objects. Refer to 6.30 for details about this function.

3.3 Getting filesystem root object

The filesystem registers itself into Device Manager (**SCP_devman** process) with name **sdd_chanfs**. This name must be used in order to get filesystem root object for further operations.

The name **sdd_chanfs** may be changed by modifying an appropriate configuration option (refer to 3.13.6.1.1).

There are two possibilities to get the filesystem root object, depending on a function used.

Getting filesystem root using **sdd_*** functions, which are part of SCIOPTA SDD:

```
sdd_obj_t *man;
sdd_obj_t *chanFs;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
chanFs = sdd_manGetByName(man, "sdd_chanfs");
```

Getting filesystem root using **sfs_get** (6.30) function:

```
sdd_obj_t *chanFs;

chanFs = sfs_get(NULL, "/sdd_chanfs", SFS_ATTR_ROOT);
```

3.4 Path formats and filesystem tree

Every object in the filesystem can be accessed using a path. The objects can be:

- filesystem root
- physical drives
- directories
- files

The path can be either absolute or relative to the current object.

3.4.1 Absolute path

Absolute path is a format which must be used in following functions if reference object is not specified (=NULL). The absolute format cannot be used in any other function.

- sfs_create (6.6)
- sfs_delete (6.7)
- sfs_deleteRecursive (6.8)
- sfs_get (6.30)
- sfs_fopen (6.17)

The format of an absolute path is shown below:

```
/sdd_chanfs/<physical_drive_name>/p1/directory/file
```

Every absolute path starts with '/' symbol which represents the Device Manager process.

The root symbol is followed by **sdd_chanfs**, which is the name the filesystem uses to register itself into Device Manager.

The **sdd_chanfs** is followed by another '/' symbol, which at this point represents the **filesystem root**.

The filesystem root is followed by a name of a physical drive which was added to the filesystem.

Physical drive name is followed by a partition number. Physical drive can have one FAT filesystem occupying the entire drive (accessible with name **p1**), or drive can be partitioned into maximum of four partitions, each containing FAT filesystem (accessible using names **p1-p4**).

The partition name is followed by one or more directories and optional file name at the end of the path, all separated by '/' symbol.

3.4.2 Relative paths

The filesystem allows using relative paths. The relativeness is always against the current object (physical drive, partition or a directory), which is passed to function.

The example below shows a path which is relative to current object which is a filesystem root. The leading '/' represents a filesystem root object. Meanings of remaining parts are the same as in case of absolute path.

```
/<physical_drive_name>/p1/directory/file
```

Leading **'/'** is optional and may be omitted in previous example. The example below shows a path which is relative to current object which is a physical drive.

```
p1/directory/file
```

3.4.3 Upper directory

Symbol '**..**' can be used to access an upper directory, where upper directory can be a directory, partition, drive or a filesystem root.

The example below shows a path with '**..**' symbol. The path, when resolved, represents a filesystem root.

```
/<physical_drive_name>/p1/directory/../../..
```

3.4.4 Filesystem tree

The filesystem root is a directory which contains added physical drive. Each physical drive contains FAT filesystems (either entire drive or partitions) with names **p1..p4**. Each FAT filesystem contains a tree of directories and files.

3.5 Adding physical drives to filesystem tree

Before a physical drive can be accessed through the filesystem, it must be added to the filesystem tree.

To add a physical drive to the filesystem tree, the **sfs_assign** (6.2) function must be used. Refer to specified chapter for details about parameters and usage.

When physical drive is added to the filesystem, a **per-physical-drive-process** is assigned to it. The process forms a layer between user application and physical drive.

After the **per-physical-drive-process** is created, the physical drive is checked for existing filesystems. All filesystems are mounted automatically and become available with names **p1-p4**. This feature is called **automounter** and can be switched-off (3.13.6.3.3).

A physical drive can be assigned to filesystem in read-only mode, by specifying appropriate option to **sfs_assign** function call (6.2.3).

When physical drive is assigned to the filesystem, it is opened by the filesystem and remains opened until it is removed (for removing a physical drive from the filesystem, refer to 3.6). During the time a physical drive is added to the filesystem it will not be possible (depending on the sharing policy of the physical device driver) to perform any operations on the device from other processes.

3.6 Removing physical drives from the filesystem tree

To remove a physical drive from the filesystem tree, the **sfs_unassign** (6.54) function must be used. Refer to specified chapter for details about parameters and usage.

Physical drive can be removed in non-forced or forced way.

When a physical drive is removed from the filesystem tree, the **per-physical-drive-process** is killed or re-started, depending on a policy used for **per-physical-drive-processes**. Refer to 3.13.2.4 for details about per-physical-drive-processes policy.

3.6.1 Non-forced drive removal

Before physical drive can removed from the filesystem tree, following necessary steps must be taken:

- All files must be closed and file objects must be freed (3.8.8, 3.10).
- All directory objects must be freed (3.10).
- All FAT filesystems must be unmounted (3.7.6).
- All raw accesses to the physical drive or partitions must be closed (3.8.8).
- All object representing a physical drive or a partition must be freed (3.10).

If above steps were not taken prior to calling **sfs_unassign** function, **EBUSY** error will be returned. Refer to 6.54.5 for complete list of errors.

3.6.2 Forced drive removal

Drive can be removed in a forced way if it is required by application. Refer to 6.54.3 for details on how to re-quest forced drive removal.

If there are objects allocated which are associated with drive being removed, the **sfs_unassign** returns **EFSOBJECTSNOTFREED** error, indicating that forced removal is initialized, but application must free objects first to finish the removal.

If for any reason objects cannot be freed, it is possible to force objects freeing. This can be done either immediately at the first forced drive removal request, or later, by calling **sfs_unassign** function again with appropriate parameters. Refer to 6.54.3 for details on how to request forced drive removal in combination with forced objects removal.

If objects are freed in a forced way, it is still required to free all sdd_obj_t objects which remain on the application side. This can be done by calling following function:

```
sc_msgFree((sc_msgptr_t)&object);
```

where **object** is of type:

```
sdd_obj_t *
```

Following is an example of removing physical drive from the filesystem in a forced way.

```
if (sfs_unassign(fileSystem, "ramdisk", 1, 0) < 0) {

  if (sc_miscErrnoGet() == EFSOBJECTSNOTFREED) {

    /*
     * sfs_unassign failed becuase of allocated objects.
     * wait 5000 ms until user application frees objects.
     */

    if (sfs_waitRm(fileSystem, "ramdisk", sc_tickMs2Tick(5000)) < 0) {

      if (sc_miscErrnoGet() == ETIMEDOUT) {

        /*
         * Application failed cleaning up the objects. Kill application process.
         */

        sc_procKill(user_hello_pid, 0);

        /*
         * Unassign drive again, this time with forced objects freeing.
         */

        if (sfs_unassign(fileSystem, "ramdisk", 1, 1) < 0) {
          failure();
        }

        /*
         * At this point physical drive process is killed (restarted) and drive
         * is completely removed from the filesystem.
         */

      } else {
        failure();
      }

    } else {
      /* Application cleaned up the objects */
    }

  } else {
    failure();
  }
}
```

3.7 Working with drives and partitions

3.7.1 Getting physical drive object

To get a physical drive object, use **sfs_get** (6.30) function with the filesystem root object and physical drive name as parameters. Refer to specified chapter for details about parameters and usage.

Following is an example of getting an object representing a ram disk block device, which was previously added to the filesystem (refer to 3.5 for details about adding a physical drive to the filesystem).

```
ramdisk = sfs_get(filesystemRootObject, "/ramdisk0", SFS_ATTR_DRIVE);
```

3.7.2 Listing all physical drives added to the filesystem

To list all physical drives added to the filesystem use functions **sfs_getFirst** (6.32) and **sfs_getNext** (6.33). Listed directory object must be a filesystem root (3.3). Refer to specified chapter for details about parameters and usage of each function.

3.7.3 Partitioning physical drives

A physical drive can be divided into maximum of four partitions.

To partition a physical drive use function **sfs_fdisk** (6.10). The function allows to specify sizes for each partition. Refer to specified chapter for details about parameters and usage.

3.7.4 Creating FAT filesystem

A FAT filesystem can occupy entire physical drive, or can be placed on any of maximum four partitions on a partitioned drive. A FAT filesystem can be created using **sfs_format** (6.18) function. Refer to specified chapter for details about parameters and usage.

3.7.5 Mounting FAT filesystem

Before the content of a FAT filesystem can be accessed (directories, files), a drive or a partition must be mounted. Mounting happens automatically when physical drive is added to the filesystem tree (automounter, 3.13.6.3.3), or can be done manually at any point (for example after formatting a partition).

A FAT filesystem on a drive or on a partition can be mounted using **sfs_mount** (6.37) function. Refer to specified chapter for details about parameters and usage.

3.7.6 Unmounting FAT filesystem

After user application finishes working with FAT filesystem, it must be unmounted by using **sfs_umount** (6.53) function. Refer to specified chapter for details about parameters and usage.

3.7.7 Getting mounted physical drive or partition object

Mounted FAT filesystems are available under names **p1..p4**. Each name means one of maximum four partitions. In case the physical drive is not divided into partitions, and FAT filesystem occupies entire drive, name **p1** must be used.

To get mounted FAT filesystem object, use **sfs_get** (6.30) function with appropriate reference object and path, as parameters. Refer to specified chapter for details about parameters and usage.

Following are examples of getting an object representing a second partition on the ram disk block device, which was previously added to the filesystem (refer to 3.5 for details about adding a physical drive to the filesystem).

Example of getting a partition object when filesystem root object is a reference:

```
partition = sfs_get(filesystemRootObject, "/ramdisk0/p2", SFS_ATTR_PARTITION);
```

Example of getting a partition object when ramdisk object is a reference:

```
partition = sfs_get(ramdiskObject, "/p2", SFS_ATTR_PARTITION);
```

3.7.8 Listing mounted FAT filesystems of a physical drive

To list all mounted FAT filesystems on a specific physical drive, use functions **sfs_getFirst** (6.32) and **sfs_getNext** (6.33) with physical drive object specified as an object to be listed. Refer to specified chapter for details about parameters and usage of each function.

Following is an example of listing all mounted FAT filesystems on a ramdisk object.

partition->name holds partition names in range **p1..p4**.

```
partition = sfs_getFirst(ramdisk);
while (partition) {
  logd_printf(logd, LOGD_INFO, "Partition: %s \n", partition->name);

  previous = partition;
  partition = sfs_getNext(ramdisk, previous);
  if (sfs_free(&previous) < 0) {
    failure();
  }
}
```

3.7.9 Monitoring drives and partitions

The filesystem allows to request a notification about following events:

- when physical drive is added or removed from the filesystem tree
- when a drive or a partition is mounted or unmounted

To request a notification use functions **sfs_waitAdd** (6.55) or **sfs_waitRm** (6.56). Refer to specified chapter for details about parameters and usage of each function.

3.7.10 Getting size of drive or partition

To get a size of a physical drive or a partition object use functions:

- for sizes less than 4GB: **sfs_sizeTotal** (6.49), **sfs_sizeUsed** (6.50), **sfs_sizeFree** (6.48), **sfs_sizeBad** (6.47)
- for sizes of 4GB or more: **sfs_size64Total** (6.45), **sfs_size64Used** (6.46), **sfs_size64Free** (6.44), **sfs_size64Bad** (6.43)

Refer to specified chapter for details about parameters and usage of each function.

3.7.11 Partition label and serial number

To get a label and a serial number of a partition use **sfs_getProperty** (6.34) function.

To set a partition label use **sfs_setProperty** (6.42) function.

Refer to specified chapter for details about parameters and usage of each function.

3.7.12 Accessing drives and partitions in raw mode

Entire drive or a specific partition can be accessed for read and write in a mode called "**raw**". This mode allows an unlimited access to every single byte located on a physical drive or a partition, regardless whether there is a valid FAT filesystem or not.

To get a "raw"-access-capable object representing either a physical drive or a partition, "raw" prefix needs to be added to the normal name of a physical drive or a partition, when using **sfs_get** (6.30) function. The prefix may be changed by setting the configuration option (3.13.6.1.4, 3.13.6.1.5)

Following is an example of getting an object representing a physical drive, which is normally available under name **ramdisk0**:

```
ramdisk = sfs_get(filesystemRootObject, "/rawramdisk0", SFS_ATTR_FILE);
```

Following is an example of getting an object representing a second partition on drive **ramdisk0**.

```
rawpartition = sfs_get(ramdisk0Object, "/rawp2", SFS_ATTR_FILE);
```

Unlike in case of mounted FAT filesystems, where name **p1** could represent a FAT filesystem occupying entire drive (3.7.7), names **rawp1..rawp4** can be used only if drive is divided into partitions.

The object representing a raw access can be further used as if it was a file. Following operations are possible:

- Copying file (3.8.3)
- Opening file for reading and writing (3.8.7)
- Closing file (3.8.8)
- Reading from file (3.8.9). The read size must a multiple of physical drive sector size.
- Writing to file (3.8.10). The write size must be a multiple of physical drive sector size.
- Moving file position marker (3.8.11). Offset must be aligned to physical drive sector size.
- Getting file end mark (3.8.12)
- Getting file size (3.8.13)

3.8 Working with files

3.8.1 Creating file

To create a new file, use function **sfs_create** (6.6).

3.8.2 Deleting file

To delete a file, use function **sfs_delete** (6.7).

3.8.3 Copying files

Single files can be copied across all physical drives and partitions. To copy a file from one location to another, use function **sfs_copy** (6.5).

3.8.4 Moving files

Files can be moved across all physical drives and partitions. However, if file is to be moved within the same partition, a faster function is available, which does not move the content of the file.

- Use function **sfs_move** (6.38), to move a file within the same partition.
- Use function **sfs_move2** (6.39), to move a file across all physical drives and partitions.

3.8.5 Getting file object

To perform certain operations on a file, such as getting or setting file attributes, it is required to get a file object from the filesystem. To get an object for a specific file, use **sfs_get** (6.30) function.

To get all files in a directory, use functions **sfs_getFirst** (6.32) and **sfs_getNext** (6.33).

To get all files in a directory which match specified pattern, use functions **sfs_findFirst** (6.15) and **sfs_findNext** (6.16).

3.8.6 Getting current path of a file object

To get a current working directory of a file (effectively a directory which contains the file), use function **sfs_getcwd** (6.31). Path returned is a relative path and starts from the filesystem root.

3.8.7 Opening file for reading and writing

File can be opened for reading/writing in two ways:

- If file object is already retrieved, use **sfs_open** (6.41) to open the file.
- To open a file using path only, use **sfs_fopen** (6.17).

3.8.8 Closing file

After operations on a file are finished, and file is no longer needed, it must be closed. To close a file, use function **sfs_fclose** (6.9).

3.8.9 Reading from file

Data can be read from file in binary and text mode, depending on a used function.

- To read data from a file in binary mode, use **sfs_fread** (6.22).
- To read single line of text (end line characters are interpreted), use function **sfs_fgets** (6.14).

3.8.10 Writing to file

Data can be written to a file in binary or text mode, depending on a used function.

- To write data to a file in binary mode, use **sfs_fwrite** (6.29).
- To write text data (end line characters are interpreted) to a file, use function **sfs_fputs** (6.20).
- To write a single character to a file (end line character is interpreted), use function **sfs_fputc** (6.19).

3.8.11 Moving file position pointer

To move the file position where data is read from or written to, file seek functions can be used.

- For offsets in range from (-2GB) to (2GB-1), use function **sfs_fseek** (6.25).
- For any offset, use function **sfs_fseek64** (6.26).

3.8.12 File error and end-of-file indicators

Each opened file object is associated with error indicator and end-of-file indicator. Indicators are stored inside an **sdd_obj_t** structure which represents a file object. This structure is a message owned by the process which uses this file object.

Whenever error indicator is set, process **errno** variable is also set to the same value.

To get error indicator of a file object, use function **sfs_ferror** (6.12). In case error indicator of a file object is not set, process **errno** variable will be returned. If file object is NULL, **EFSNULLPOINTER** error will be returned and process **errno** variable will also be set to **EFSNULLPOINTER**.

To get end-of-file indicator of a file, use function **sfs_feof** (6.11). In case file object passed is NULL, process **errno** variable will be set to **EFSNULLPOINTER**.

Error indicator, end-of-file indicator and process **errno** variable are sticky flags and can only be cleared by following:

- To clear error indicator, end-of-file indicator and process **errno** variable altogether, use function **sfs_clearerr** (6.3). In case file object passed is NULL, process **errno** variable will be set to **EFSNULLPOINTER**.
- In addition to **sfs_clearerr**, end-of-file indicator is cleared by a successful call to sfs_fseek (6.25), sfs_fseek64 (6.26), sfs_fresize (6.23), sfs_fresize64 (6.24).

Functions listed below take file object as a parameter. It means that if a file object passed to these functions is NULL, error indicator cannot be set, and only process **errno** variable will be set to **EFSNULLPOINTER**.

Following functions may set **error indicator** in case of an error (for a non-NULL file object):

- sfs_fread (6.22)

- sfs_fwrite (6.29)
- sfs_fgets (6.14)
- sfs_fputs (6.20)
- sfs_fputc (6.19)
- sfs_fseek (6.25)
- sfs_fseek64 (6.26)
- sfs_fresize (6.23)
- sfs_fresize64 (6.24)
- sfs_fflush (6.13)

Following functions may set **end-of-file indicator** in case of trying to read at the end of file (for a non-NULL file object):

- sfs_fread (6.22)
- sfs_fgets (6.14)

Following code example summarizes rules of using error indicator and end-of-file indicator to assure errors are correctly recognized.

```
sfs_clearerr(file); //clear error indicator and end-of-file indicator

if (sfs_ferror(file) == 0) {

    if (sfs_fread(buffer, 1, sizeof(buffer), file) == sizeof(buffer))
    {
        // all data successfully read from file
    }
    else
    {
        if (sfs_feof(file) != 0)
        {
            // end of file encountered
        }
        else
        {
            error = sfs_ferror(file);

            if (error == EFSNULLPOINTER)
            {
                // NULL pointer was specified to either
                // sfs_fread, sfs_feof or sfs_ferror
            }
            else
            {
                // other error
            }
        }
    }
}
else
{
    // NULL pointer was specified to either sfs_clearerr or sfs_ferror
}
```

3.8.13 Getting file size

To get the size of a file object use functions **sfs_sizeTotal** (6.49), **sfs_sizeUsed** (6.50), **sfs_size64Total** (6.45) or **sfs_size64Used** (6.46).

3.8.14 Resizing file

A file can be either expanded to maximum supported limit, which is 4GB-1 bytes, or can be shrunk down to 0 bytes. To change size of a file in range from 0 to 2GB-1 bytes, use function **sfs_fresize** (6.23). For larger range from 0 to 4GB-1, use function **sfs_fresize64** (6.24).

3.8.15 File attributes

FAT filesystem supports following file attributes: read-only, archive, system and hidden. To get or set attributes of a file, use functions **sfs_getProperty** (6.34) and **sfs_setProperty** (6.42), respectively.

3.8.16 File modification time

FAT filesystem stores last modification time of a file. To get or set last modification time, use functions **sfs_timeGet** (6.51) and **sfs_timeSet** (6.52), respectively.

3.8.17 File cache size

Filesystem allocates a read/write cache for each opened files. The goal of using file cache is to minimize number of writes to physical drive by merging sectors into blocks, which are usually bigger than sectors. It is especially important for flash devices (including SD cards), where a block has to erased before it can be written.

Default cache size is equal to one physical drive sector size. This is also a minimal cache size.

When changing cache size, first step is to allocate a new cache before the old one is freed.

Before the old cache is freed, entire cache content is flushed to the physical drive.

To change cache size, use function **sfs_setProperty** (6.42).

To get current cache size, use function **sfs_getProperty** (6.34).

3.8.18 File cache merge buffer

Each opened FAT file uses cache to minimize the number of writes to physical medium. Refer to 3.8.17 for details about file cache size.

When file is synced from cache to physical medium, it is important from the performance point of view to write data in blocks as large as possible, up to the block size of a physical drive.

A block is a multiple of sectors and is larger than sector. When file cache is flushed, cached sectors may not be contiguous in memory, which means they have to be reordered. If merge buffer is disabled, sectors are reordered by swapping cached sectors, which requires 3 memory copy operations, at worst. If merge buffer is enabled, each sector is copied to the merge buffer and the merger buffer will contain contiguous sectors - it will require only one memory copy operation per sector.

The best performance is achieved when merge buffer size is equal to the block size of physical drive, however, the block size may be too big in terms of memory constraints. The general rule is that merge buffer should be as big as the system memory allows (up to the block size).

A physical drive (like nand flash) may report the size of a block, by implementing IOCTL command BLK-

DEVGETERASEBLKSIZE.

The merge buffer is allocated once per physical drive, regardless of the number of opened files.

Refer to 3.13.6.5.2 for informations about enabling merge buffer.

3.8.19 Getting file name in DOS 8.3 format

To get a file name in a short DOS 8.3 format, use **sfs_getShortName** (6.35).

3.9 Working with directories

3.9.1 Creating directory

To create a new directory, use function **sfs_create** (6.6).

3.9.2 Deleting directory

To delete a directory use following functions:

- if directory is empty, use function **sfs_delete** (6.7).
- if directory is either empty or not empty, use function **sfs_deleteRecursive** (6.8).

3.9.3 Getting directory object

To perform certain operations on a directory, such as getting or setting directory attributes, it is required to get a directory object from the filesystem. To get an object for a specific directory, use **sfs_get** (6.30) function.

To get all directories from a specific directory, use functions **sfs_getFirst** (6.32) and **sfs_getNext** (6.33).

To get all directories which match specified pattern, from a specified directory, use functions **sfs_findFirst** (6.15) and **sfs_findNext** (6.16).

3.9.4 Listing all items in a directory

To list all items in a directory (subdirectories and files) use functions **sfs_getFirst** (6.32) and **sfs_getNext** (6.33).

3.9.5 Listing items in a directory using search pattern

To get all subdirectories and files which match specified pattern, from a specified directory, use functions **sfs_findFirst** (6.15) and **sfs_findNext** (6.16).

3.9.6 Copying directory

A directory with its content can copied across all physical drives and partitions. To copy a directory from one location to another, use function **sfs_copy** (6.5).

3.9.7 Moving directory

Directory with its content can be moved across all physical drives and partitions. However, if a directory is to be moved within the same partition, a faster function is available, which does not move the content of the directory.

- Use function **sfs_move** (6.38), to move a directory within the same partition.
- Use function **sfs_move2** (6.39), to move a directory across all physical drives and partitions.

3.9.8 Getting current path of a directory object

To get a full path of a directory, starting from the filesystem root (path relative to the filesystem root), use function **sfs_getcwd** (6.31).

3.9.9 Changing current path of a directory object

To change the path (current working directory) of a directory object, use function **sfs_chdir** (6.4).

3.9.10 Getting directory size

To get the size of a directory object, use functions **sfs_sizeTotal** (6.49), **sfs_sizeUsed** (6.50), **sfs_sizeFree** (6.48), **sfs_size64Total** (6.45), **sfs_size64Used** (6.46), **sfs_size64Free** (6.44).

3.9.11 Directory attributes

FAT filesystem supports following file attributes: read-only, archive, system and hidden. To get or set attributes of a directory, use functions **sfs_getProperty** (6.34) and **sfs_setProperty** (6.42), respectively.

3.9.12 Directory modification time

FAT filesystem stores last modification time of a directory. To get or set the last modification time, use functions **sfs_timeGet** (6.51) and **sfs_timeSet** (6.52), respectively.

3.9.13 Getting directory name in DOS 8.3 format

To get directory name in short DOS 8.3 format, use **sfs_getShortName** (6.35).

3.10 Freeing filesystem objects

The filesystem keeps track of all allocated object, which may represent a filesystem root, physical drive, partition (FAT filesystem) , directory or a file. Every object must be freed when it's no longer used. The list below contains all the functions returning a newly allocated object, which must be freed.

- sfs_get (6.30)
- sfs_getFirst (6.32)
- sfs_getNext (6.33)
- sfs_findFirst (6.15)
- sfs_findNext (6.16)
- sfs_fopen (6.17)

Not freeing objects may result with **EBUSY** error when partition is unmounted or physical drive removed from the filesystem. In case of an error, user application must free all the remaining objects and retry the operation again.

Each physical drive object supports a debug feature, which allows the user application to retrieve the list of allocated objects for a particular physical drive object. Refer to 6.34.4.5 for details about using this feature.

3.11 Accidental drive removal

In the normal case, if user application wants to remove physical drive from the filesystem, it uses **sfs_unassign** function (6.54). Prior to the function call, user application must close all the files, free all filesystem objects (refer to 3.10 for details about freeing objects), and unmount all mounted FAT filesystems on that drive (refer to 3.7.6 for details about unmounting FAT filesystems). Application can also remove physical drive from filesystem in a forced way. Refer to 3.6 for details about removing drives from the filesystem.

The **per-physical-drive-process** observes the physical drive for the condition of accidental removal.

In case the physical drive is removed accidentally, the **per-physical-drive-process** switches to a special mode where almost all operations return **EFSDRIVEREMOVED** error. The **per-physical-drive-process** stays alive until all objects associated with the physical drive are freed.

In a well-written user application, every operation should be checked for a possible error. This includes **EFSDRIVEREMOVED** error. User application should react to it in an appropriate way described above, to allow **per-physical-drive-process** to die.

If, for some reason, it is impossible for the application to gracefully clean-up when drive is accidentally removed, the **sfs_unassign** function can be used in the same way as if it was used to force drive removal with forced objects freeing. For details refer to 3.6.2. However, it is still required to free all sdd_obj_t objects which remain on the application side. This can be done by calling following function:

```
sc_msgFree((sc_msgptr_t)&object);
```

where **object** is of type:

sdd_obj_t *

3.12 Filesystem error hook

The filesystem requires an error hook to be defined by the user. This error hook will be called when fatal error inside filesystem is detected and filesystem cannot be used anymore.

The error hook is a function which must not return.

The declaration of filesystem error hook is located in file:

<installation_folder>\sciopta\<version>\include\fat\chanFs.h

and is declared as:

```
extern void chanfs_error_hook(uint32_t code);
```

The error hook gives the user a possibility to write their own error handling code to react to filesystem failure.

3.13 Configuration

3.13.1 Introduction

Filesystem configuration depends on a specific requirements of the system design. The requirements include:

- Per-physical-drive-processes created dynamically or defined as static processes
- Automounting available FAT filesystems on physical drives
- Support for large drive
- Cache for files accesses
- FAT structures cache
- Support for multi-partitions
- Support for large drives (> 4GB)
- Support for long file names
- Support for creating FAT filesystem on a blank drive/partition

3.13.2 SCONF Configuration

3.13.2.1 Introduction

Start the SCIOPTA configuration utility (sconf.exe) and define and configure all static modules, processes and message pools of your system. Please see the SCIOPTA - Target Manual for your selected processor for more information about using sconf.exe and the configuration process. All static objects will be generated and started automatically at system start.

3.13.2.2 Message pool

To use a separated pool for the filesystem, define a pool with name "**fs_pool**" in the same module as filesystem root process (3.13.2.3).

3.13.2.3 Filesystem root process

This is a prioritized process which represents the filesystem root and holds a collection of physical drives added to the filesystem.

Priority Process Name	SCP_chanFs
Priority Process Function	SCP_chanFs
	This is the address where the created process will start execution.
Stack Size	The exact stack size is target processor, compiler and system dependent.
	A good starting point is to use the values of the delivered examples. Use a process stack analysing tool such as the SCIOPTA DRUID system level debugger to optimize the stack size.
Priority	Depending on the priority design and the real-time requirements of the whole

	system. The priority of this process should be less than the priority of physical devices drivers which are added to the filesystem.
Process State	Started.

3.13.2.4 Per-physical-drive-processes

3.13.2.4.1 Introduction

The per-physical-drive-processes may be created dynamically by the filesystem root process or a number of such processes may be defined statically in the system.

3.13.2.4.2 Dynamically created processes

If filesystem is configured for dynamic processes (CHANFS_USE_DYNAMIC_PROCESSES=1, refer to 3.13.6 for details) , the per-physical-drive-process is created when physical drive is added to the filesystem. User does not need to defined any per-physical-drive-process in this case.

The name of the newly created process is temporary and uses following format:

SCP_chanFsPhys<random number>

The stack size is set to CHANFS_PHYSICAL_DRIVE_PROCESS_STACK_SIZE configuration option (3.13.6).

3.13.2.4.3 Statically defined processes

If filesystem is configured for static processes (CHANFS_USE_DYNAMIC_PROCESSES=0, refer to 3.13.6 for details) , the filesystem root process expects the per-physical-drive-processes to be defined in the same module as the filesystem root process (3.13.2.3), and started. The filesystem chooses an available process to be used with physical drive, which is being added to the filesystem.

The number of static per-physical-drive-processes depends on a configuration for multi-partitions and number of supported volumes (FAT filesystems). Refer to 3.13.6 for details about configuring multi-partitioning and number of volumes.

The formula for calculating the required number of static per-physical-drive-processes is as follows:

Multi-partition support is enabled (MULTI_PARTITION=1):

```
number_of_processes = ((VOLUMES / 4)+((VOLUMES % 4)?(1):(0)))
```

Multi-partition support is disabled (MULTI_PARTITION=0):

```
number_of_processes = VOLUMES
```

Both **VOLUMES** and **MULTI_PARTITION** are filesystem configuration options. Refer to 3.13.6 for details.

Per-physical-drive-process is a prioritized process which represents a physical drive added to the filesystem and must be configured as follows, if static processes configuration is selected.

Priority Process Name	SCP_chanFsPhysX
	X is a zero-based index of a per-physical-drive-process. The X is in decimal format: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, … , <number_of_required_processes - 1>

Priority Process Function	SCP_chanFsPhys
	This is the address where the created process will start execution.
Stack Size	The exact stack size is target processor, compiler and system dependent.
	A good starting point is to use the values of the delivered examples. Use a process stack analysing tool such as the SCIOPTA DRUID system level debugger to optimize the stack size.
Priority	Depending on the priority design and the real-time requirements of the whole system.
	The priority of this process should be less than the priority of the filesystem root process.
Process State	Started.

3.13.3 Recommended setup

3.13.3.1 Introduction

In order for the filesystem processes to start and communicate correctly there is a specific priority order that needs to be kept when defining filesystem processes.

3.13.3.2 Dynamically created processes

For dynamic processes configuration, only one static process must be defined, which is a filesystem root process. The following priority relations must be kept in the system:

PRIORITY(**PHYSICAL DEVICE DRIVERS**) > PRIORITY(**FILESYSTEM ROOT PROCESS) >** PRIORITY(**PER-PHYSICAL-DRIVE PROCESSES**) > PRIORITY(**USER APPLICATION**)

The filesystem root static process creates per-physical-drive processes with priorities of PRIORITY(**FILESYSTEM ROOT PROCESS) minus 1**. Thus, filesystem root static process priority must be set to a value of at least 30 (assuming there is no user application using filesystem) or higher.

3.13.3.3 Statically created processes

For static processes configuration, two or more filesystem-specific static processes must be defined (one filesystem root process and a non-zero number of per-physical-drive-processes). The following priority relations must be kept in the system:

PRIORITY(**PHYSICAL DEVICE DRIVERS**) > PRIORITY(**FILESYSTEM ROOT PROCESS**) > PRIORITY(**PER-PHYSICAL-DRIVE PROCESS**) > PRIORITY(**USER APPLICATION**)

3.13.4 Configuring for minimal memory usage

To configure FAT filesystem for minimal memory usage, set following setting:

CHANFS_FILE_CACHE_MERGE_BUFFER_MAX_SIZE = 0 (refer to 3.13.6.5.2)

FAT_CACHE_DEFAULT_SIZE = 0 (refer to 3.13.6.5.1)

USE_LFN = 0 (refer to 3.13.6.4.4)

VOLUMES = 1 (refer to 3.13.6.2.2)

MAX_SS = maximum sector size, which will be used in the target (usually 512 B, refer to 3.13.6.2.5)

FS_LOCK = 1 (refer to 3.13.6.2.3)

CHANFS_PHYSICAL_DRIVE_PROCESS_STACK_SIZE = value adjusted for the specific target (refer to 3.13.6.3.1

3.13.5 Configuring for maximum performance

To configure FAT filesystem for maximum performance, set following setting:

CHANFS_FILE_CACHE_MERGE_BUFFER_MAX_SIZE = block size (refer to 3.13.6.5.2)

FAT_CACHE_DEFAULT_SIZE = maximum value allowed by the target memory size (refer to 3.13.6.5.1)

3.13.6 Configuration options reference

3.13.6.1 Names

3.13.6.1.1 Filesystem registration name

Option name:	CHANFS_FS_NAME
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Valid SCIOPTA SDD object name
Description:	This is the name the filesystem root process uses to register itself into the device manager.

3.13.6.1.2 Filesystem root process name

Option name:	CHANFS_MAIN_PROCESS_NAME
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Valid SCIOPTA process name
Description:	This is the name of the filesystem root process.

3.13.6.1.3 Per-physical-drive-process name

Option name:	CHANFS_PHYSICAL_PROCESS_NAME
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Alfa-numeric string
Description:	This is the name of per-physical-drive-processes. There is a random decimal number added to this name (for dynamic process creation mode) or decimal number in range 0..N-1 (for static processes declaration). Refer to 3.13.2.4 for details about per-physical-drive-processes creation/declaration.

3.13.6.1.4 Partition name prefix

Option name:	CHANFS_PART_NAME_PREFIX
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Alfa-numeric string
Description:	This is the prefix which will be used to form a partition name. The prefix will be followed by a number in range 1..4

3.13.6.1.5 Drive name prefix for raw access

Option name:	CHANFS_DISK_NAME_RAW_PREFIX
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Alfa-numeric string
Description:	This is the prefix which will be used to form a drive name, which is intended for raw accessing. For details about raw mode refer to 3.7.12.

3.13.6.1.6 Partition name prefix for raw access

Option name:	CHANFS_PART_NAME_RAW_PREFIX
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Alfa-numeric string
Description:	This is the prefix which will be used to form a partition name, which is intended for raw accessing. The prefix will be followed by a number in range 1..4. For details about raw mode refer to 3.7.12.

3.13.6.2 Capabilities

3.13.6.2.1 Multi partition support

Option name:	MULTI_PARTITION
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables support for multi-partition physical drives.

3.13.6.2.2 Maximum number of supported FAT filesystems

Option name:	VOLUMES
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1, N
Description:	This option configures the number of FAT filesystems which can be used simultaneously.

If multi-partition support is disabled, the VOLUMES is equal to the number of supported physical drives.

If multi-partition support is enabled and:

- VOLUMES is a multiple of 4: the number of supported drives is VOLUMES divided by 4. All partitions on each drive can be mounted simultaneously.
- VOLUMES is not a multiple of 4: the number of supported drives is VOLUMES rounded up to multiple of 4 and then divided by 4. However, only first VOLUMES partitions can be mounted – last partition(s) on a last drive will not be accessible.

Below examples are with assumption that MULTI_PARTITION option is enabled.

Example 1: First drive is divided into 4 partitions and second drive is also divided into 4 partitions. To mount all the partitions, set VOLUMES=8

Example 2: First drive is not divided (FAT filesystem occupies entire drive) and second drive is divided into 4 partitions. To mount all the filesystems, set VOLUMES=8. This is because each physical drive (except the last one) takes 4 volumes, even if it's not divided into partitions.

Example 3: This example is similar to example 2, but drives order is swapped. First drive is divided into four partitions and second drive is not divided (FAT filesystem occupies entire drive). To mount all the filesystems, set VOLUMES=5 (4 volumes for first drive, 1 volume for second drive).

Example 1: First drive is divided into 4 partitions and second drive is also divided into 4 partitions, but only first 3 of the second drive will be mounted (7 in total on both drives). To mount all the partitions it is enough to set VOLUMES=7.

3.13.6.2.3 Maximum number of simultaneously opened files and directories

Option name:	FS_LOCK
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (for read-only configuration) 1..N (for read-write configuration)
Description:	Number of files and directories that can be opened simultaneously per physical drive.

Files and directories are protected by locks against simultaneous accesses.

When file is opened, one lock is taken from the pool of locks. When file is closed, lock is returned to the pool of locks.

A lock is taken from the pool of locks when one of following functions or messages is used against a directory object:

- sfs_copy (6.5)
- sfs_deleteRecursive (6.8)
- sfs_findFirst (6.15)
- sfs_findNext (6.16)
- sfs_getFirst (6.32)
- sfs_getNext (6.33)
- sfs_move2 (6.39)
- SDD_MAN_GET_FIRST (5.14)
- SDD_MAN_GET_NEXT (5.15)
- SFS_FFIRST (5.30)
- SFS_FNEXT (5.31)

When directory object is freed (refer to 3.10), lock is returned to the pool of locks.

3.13.6.2.4 Support for large drives

Option name:	CHANFS_SUPPORT_FOR_LARGE_DRIVES
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables support for large drives (with size > 4GB)

3.13.6.2.5 Sector size

Option name:	MAX_SS
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	512, 1024, 2048, 4096
Description:	Maximum supported sector size.

Option name:	MIN_SS
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	512, 1024, 2048, 4096
Description:	Minimum supported sector size.

3.13.6.3 Behaviour

3.13.6.3.1 Per-physical-drive-process stack size

Option name:	CHANFS_PHYSICAL_DRIVE_PROCESS_STACK_SIZE
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h

Valid values:	The exact stacksize is target processor, compiler and system dependent.
Description:	A good starting point is to use the values of the delivered examples. Use a process stack analysing tool such as the SCIOPTA DRUID system level debugger to optimize the stack size.

3.13.6.3.2 Per-physical-drive-process creation

Option name:	CHANFS_USE_DYNAMIC_PROCESSES
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This options enables mode, where per-physical-drive-processes are created dynamically when physical drive is added to the filesystem.

3.13.6.3.3 Automounter support

Option name:	CHANFS_USE_AUTOMOUNTER
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables automounter feature. If this option is enabled and physical drive is added to filesystem, it is scanned for existing FAT filesystems. Found FAT filesystems are mounted.

3.13.6.3.4 Safe automounter (only SAFE FAT product)

Option name:	CHANFS_AUTOMOUNTER_SAFE
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	If this option is enabled, automounter will mount only these existing FAT filesystems which are SAFE. If this option is disabled, automounter will try to mount SAFE FAT first and if it fails for whatever reason, then automounter will try to mount in non-SAFE mode.

3.13.6.3.5 Use Sciopta trap interface

Option name:	CHANFS_USE_TRAP_INTERFACE
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	If this option is enabled, filesystem sources will be compiled with Sciopta trap interface enabled.

3.13.6.3.6 Timeout for drive removal notifications

Option name:	CHANFS_POLLING_FOR_RM_NOTIFICATIONS_TMO_MS
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	1ms .. sc_tickTick2Ms(SC_TMO_MAX)
Description:	When a physical drive is added to the filesystem, physical drive process monitors for the drive removal by sending SDD_MAN_RM_NOTIFY message to device manager. In case the drive is removed by user with SFS_UNASSIGN message, the notification request stays in Device Manager. The timeout is specified so that the device manager could send back the reply, even to a non-existing driver process. Otherwise the request would stay in the device manager forever.

3.13.6.3.7 Waiting time for drive removal

Option name:	CHANFS_WAIT_FOR_DRIVE_REMOVAL_MS

Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	1ms .. sc_tickTick2Ms(SC_TMO_MAX)
Description:	When disk error is detected, the filesystem will wait this specified time for the device to unregister itself from the Device Manager. This wait is necessary to check whether the disk error was a consequence of drive removal. If it was, EFSDRIVEREMOVED will be returned.

3.13.6.3.8 Read-only mode

Option name:	FS_READONLY
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled),1 (Enabled)
Description:	This option allows to compile FAT filesystem in read-only mode.

3.13.6.3.9 Flash devices support

Option name:	_USE_TRIM
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled),1 (Enabled)
Description:	This option enables sectors trimming for flash devices. When enabled, the filesystem notifies underlying Flash Translation Layer that specific sectors are no longer used.

3.13.6.3.10Free space calculation and last used cluster number

Option name:	FS_NOFSINFO
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	Free space size stored in FAT structure (Bit 0): 0 (Trust), 1 (Do not trust) Last used cluster number stored in FAT structure (Bit 0): 0 (Trust), 1 (Do not trust)
Description:	If you need to know the correct free space on the FAT32 volume, set bit 0 of this option, and full FAT scan will be performed when free space size is requested for the first time. If bit 1 of this option is set, the last used cluster number, which is stored in FAT structure, will not be used.

3.13.6.4 Features

3.13.6.4.1 SAFE FAT support

Option name:	SAFE_FATFS
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables SAFE FAT support. Refer to chapter 4 for informations about SAFE FAT.

3.13.6.4.2 String functions support

Option name:	USE_STRFUNC
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled), 1 (Enabled without LF-CRLF conversion), 2 (Enabled with LF-CRLF conversion)
Description:	This option enables support for file string functions. Refer to specified header file for detailed description.

3.13.6.4.3 FAT filesystem creating support

Option name:	USE_MKFS
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables support for creating FAT filesystems on drives and partitions.

3.13.6.4.4 Long file names support

Option name:	USE_LFN
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled) or 3 (Enabled)
Description:	This option configures support for long file names. Refer to specified header file for detailed description.

Option name:	MAX_LFN
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1..N
Description:	Maximum length of a filename when long file names are enabled.

3.13.6.4.5 Characters encoding for filenames

Option name:	CODE_PAGE
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1 (ASCII - No extended character. Non-LFN configuration only) 437 (U.S.) 720 (Arabic) 737 (Greek) 771 (KBL) 775 (Baltic) 850 (Latin 1) 852 (Latin 2) 855 (Cyrillic) 857 (Turkish) 860 (Portuguese) 861 (Icelandic) 862 (Hebrew) 863 (Canadian French) 864 (Arabic) 865 (Nordic) 866 (Russian) 869 (Greek 2) 932 (Japanese (DBCS)) 936 (Simplified Chinese (DBCS)) 949 (Korean (DBCS)) 950 (Traditional Chinese (DBCS))
Description:	This option specifies the OEM code page to be used on the target system. Incorrect setting of the code page can cause a file open failure.

3.13.6.4.6 Fixed timestamps

Option name:	FS_NORTC
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0 (Disabled), 1 (Enabled)
Description:	This option enables fixed timestamps if system does not provide RTC timestamps.

3.13.6.4.7 Fixed timestamp day

Option name:	NORTC_MDAY
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1..31
Description:	Fixed timestamp day. This option is active if FS_NORTC is enabled.

3.13.6.4.8 Fixed timestamp month

Option name:	NORTC_MON
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1..12
Description:	Fixed timestamp month. This option is active if FS_NORTC is enabled.

3.13.6.4.9 Fixed timestamp year

Option name:	NORTC_YEAR
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	1980..2107
Description:	Fixed timestamp year. This option is active if FS_NORTC is enabled.

3.13.6.5 Caches

3.13.6.5.1 FAT cache default size

Option name:	FAT_CACHE_DEFAULT_SIZE
Location:	<installation_folder>\sciopta\<version>\include\fat\ffconf_template.h
Valid values:	0..N in units of drive sectors
Description:	This option configures the default size of FAT structures cache. The cache is per each mounted FAT filesystem. Refer to 6.37 for informations about specifying FAT cache size when mounting filesystem. Default cache size is used also by automounter feature. Refer to 3.13.6.3.3 for details about enabling and disabling automounter.

3.13.6.5.2 File cache merge buffer maximum size

Option name:	CHANFS_FILE_CACHE_MERGE_BUFFER_MAX_SIZE
Location:	<installation_folder>\sciopta\<version>\include\fat\chanFsConf_template.h
Valid values:	Number of bytes, 0 if disabled, power of 2 if enabled (value must be larger than sector size)
Description:	Refer to 3.8.18 for informations about file cache merge buffer.

4 SAFE FAT Filesystem

4.1 Introduction

SAFE FAT filesystem is a standard FAT filesystem, but with special log, which allows the filesystem to recover from power fail without using tools such as scandisk to repair FAT structures.

4.2 Requirements

SAFE FAT filesystem can be power-fail safe only if underlying block device is always consistent, which means that if block (sector) of memory is written by the filesystem to the memory device, it will always contain either valid old data or valid new data when power up after power fail.

For the purpose of the log, the root directory will contain **$$SFAT$$** directory, which will be inaccessible for the user when filesystem is mounted in SAFE mode.

4.3 Enabling SAFE FAT

SAFE FAT is a separated product, but it shares some of the files with non-SAFE FAT.

To enable SAFE FAT, use option SAFE_FATFS (3.13.6.4.1).

4.4 Creating SAFE FAT filesystem

SAFE FAT filesystem can be created on a partition or entire drive by creating a new SAFE FAT or by adding safety to existing non-SAFE filesystem.

4.4.1 Formatting for SAFE FAT

To create a SAFE FAT filesystem on a partition or on entire drive, use **sfs_format** function (6.18).

4.4.2 Adding safety to existing non-SAFE FAT

To add safety into an existing non-SAFE FAT partition or drive, use **sfs_ioctl** function (6.36.2) with appropriate command (6.36.4.3).

It is a responsibility of a user to make sure that existing non-SAFE filesystem is consistent, by using means such as scandisk or similar, prior to calling sfs_ioctl function. If SAFE log is created on a filesystem which is corrupted in any way, the result may be UNPREDICTABLE.

4.5 Working with directories

Except for a worse performance in compare to non-SAFE FAT (due to the log updates), working with directories is the same as in case of non-SAFE FAT.

4.6 Working with files

When SAFE FAT is used, file content is also power-fail safe. After power-fail, either old or new content of a file is available. Changes made to a file are committed only when either **sfs_fflush** (6.13) or **sfs_fclose** (6.9) function is called. When file is opened with truncate option (6.41.3, 6.17.3), change is committed immediately.

Assuring power-fail safety causes a performance drop when writing file. The level of performance drop depends on how file is written. The best performance is achieved when data is appended to the end of file. The worst performance is achieved when file is overwritten. Thus, it is recommended, whenever possible, to truncate a file before overwriting old data. Effective truncate can be achieved by either opening a file with truncate option (6.41.3, 6.17.3) or by using **sfs_fresize** / **sfs_fresize64** function (6.23 / 6.24) followed by **sfs_fflush** (6.13) function call.

4.7 Limitations

SAFE FAT allows to open up to 61 files simultaneously.

If filesystem is configured in red-only mode (3.13.6.3.8) or drive is assigned to the filesystem in read-only mode (6.2.3), and SAFE FAT filesystem experienced power-fail which requires read-write mode after power-on, the **sfs_mount** will return EFSNOTCNSTSAFEFILESYSTEM error (6.37.6).

4.8 Precautions

Mounting SAFE FAT filesystem with following assumptions:

- Mounting is non-safe (6.37.4)
- Mounting is not-read-only (6.37.4)
- Filesystem gets modified
- $$SFAT$$ directory content gets modified
- Power-fail happened prior to such non-safe mounting or happens after such non-safe mounting

may end up with corrupted filesystem if it is mounted again in SAFE FAT mode.

General rule of thumb is to avoid non-safe, not-read-only mounting.

5 Messages

5.1 Introduction

SCIOPTA is a message based real-time operating system. Interprocess communication and coordination is done by messages. Message passing is a very fast, secure, easy to use and a good to debug method.

Messages are the preferred tool for interprocess communication in SCIOPTA. SCIOPTA is specifically designed to have a very high message passing performance. Messages can also be used for interprocess coordination or synchronization duties to initiate different actions in processes. For this purposes messages can but do not need to carry data.

A message buffer (the data area of a message) can only be accessed by one process at a time which is the owner of the message. A process becomes an owner of a message when it allocates the message by the **sc_msgAlloc** system call or when it receives the message by the **sc_msgRx** system call.

Message passing is also possible between processes on different CPUs. In this case specific communication process types on each side will be needed called **SCIOPTA CONNECTOR Processes**.

In this chapter all SCIOPTA File System messages are described.

The messages are listed in alphabetical order. The request and reply message are described together.

5.1.1 Error Code

The error code is included in the error member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message error must be set to zero.

5.1.2 Header file

The messages are defined in the following header file:

<installation_folder>\sciopta\<version>\include\sfs\sfs.msg

5.2 Messages classification

5.2.1 Operations on physical drives and partitions

SDD_MAN_ADD	Add a block device to the filesystem (5.12)
SDD_MAN_GET	Get a physical drive or a partition object descriptor (5.13)
SDD_MAN_GET_FIRST	Get first physical drive or first mounted partition object descriptor (5.14)
SDD_MAN_GET_NEXT	Get next physical drive or next mounted partition object descriptor (5.15)
SDD_MAN_NOTIFY_ADD	Request a message from the filesystem root object when a physical drive is added to the filesystem or partition is mounted (5.16)
SDD_MAN_NOTIFY_RM	Request a message from the filesystem root object when a physical drive is removed from the filesystem or partition is unmounted (5.17)
SDD_MAN_RM	Remove a block device from the filesystem (5.18)
SDD_OBJ_DUP	Create a copy of an object representing a filesystem root, physical drive or a partition (5.19)
SDD_OBJ_RELEASE	Release a temporary object representing a filesystem root, physical drive or a partition (5.20)
SDD_OBJ_SIZE_GET	Get size of a physical drive or a partition (5.21)
SDD_OBJ_SIZE64_GET	Get size of a physical drive or a partition (5.22)
SDD_OBJ_TAG_GET	Get a specific property of a physical drive or a partition (5.23)
SDD_OBJ_TAG_SET	Set a specific property of a physical drive or a partition (5.24)
SFS_FDISK	Partition a physical drive (5.28)
SFS_FORMAT	Create a filesystem on a physical drive or a partition (5.33)
SFS_MOUNT	Mount a physical drive or a partition (5.39)
SFS_UMOUNT	Unmount a physical drive or a partition (5.41)

5.2.2 Operations on physical drives and partitions (raw access)

SDD_DEV_CLOSE	Close an opened disk or partition (5.3)
SDD_DEV_OPEN	Open a disk/partition object for read, write or read/write (5.5)
SDD_DEV_READ	Read data from a disk/partition (5.6)
SDD_DEV_WRITE	Write data to a disk/partition (5.7)
SDD_FILE_SEEK	Change current read/write offset of an opened disk/partition (5.10)
SDD_FILE_SEEK64	Change current read/write offset of an opened disk/partition (5.11)
SDD_MAN_GET	Get a descriptor of a physical drive or a partition for raw access (5.13)
SDD_OBJ_RELEASE	Release a temporary object representing a physical drive or a partition accessed in raw mode (5.20)
SDD_OBJ_SIZE_GET	Get size of a filesystem object: physical drive or partition (5.21)
SDD_OBJ_SIZE64_GET	Get size of a filesystem object: physical drive or partition (5.22)
SFS_FFLUSH	Flush cached information of writing to a disk/partition (5.29.1)
SFS_FTELL	Get current file position pointer (5.36)
SFS_FTELL64	Get current file position pointer (5.37)

5.2.3 Operations on FAT directories and FAT files structure

SDD_MAN_ADD	Add a new directory or a new file to an existing directory (5.12)

SDD_MAN_GET	Get a directory or a file object descriptor (5.13)
SDD_MAN_GET_FIRST	Get first directory/file in the specified directory (5.14)
SDD_MAN_GET_NEXT	Get next directory/file in the specified directory (5.15)
SDD_MAN_RM	Remove a directory or a file from an existing directory (5.18)
SDD_OBJ_DUP	Create a copy of an object representing a directory/file (5.19)
SDD_OBJ_RELEASE	Release a temporary object representing a directory/file (5.20)
SDD_OBJ_SIZE_GET	Get size of a directory/file (5.21)
SDD_OBJ_SIZE64_GET	Get size of a directory/file (5.22)
SDD_OBJ_TAG_GET	Get a specific property of a directory/file (5.23)
SDD_OBJ_TAG_SET	Set a specific property of a directory/file (5.24)
SDD_OBJ_TIME_GET	Get the last modification time of a directory/file (5.25)
SDD_OBJ_TIME_SET	Set the last modification time of a directory/file (5.26)
SFS_FFIRST	Find first object in a directory which matches the name pattern (5.30)
SFS_FNEXT	Find next object in a directory which matches the name pattern (5.31)
SFS_GETSHORTNAME	Get a short name of a directory or a file (5.38)
SFS_MOVE	Rename a directory/file and/or move it to another location within the same partition (5.40)

5.2.4 Operations on FAT files

SDD_DEV_CLOSE	Close an opened file (5.3)
SDD_DEV_OPEN	Open a file object for read, write or read/write (5.5)
SDD_DEV_READ	Read data from file (5.6)
SDD_DEV_WRITE	Write data to file (5.7)
SDD_FILE_RESIZE	Resize an opened file (5.8)
SDD_FILE_SEEK	Change current read/write offset of an opened file (5.10)
SDD_FILE_SEEK64	Change current read/write offset of an opened file (5.11)
SFS_FGETS	Get a string from a file (5.32)
SFS_FPUTC	Put a character to a file (5.34)
SFS_FPUTS	Put a string to a file (5.35)
SFS_FFLUSH	Flush cached information of writing to a file (5.29.1)
SFS_FTELL	Get a current file position pointer (5.36)
SFS_FTELL64	Get a current file position pointer (5.37)
SFS_GETSHORTNAME	Get a short name of a directory or a file (5.38)

5.3 SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY

5.3.1 Filesystem root object

This message is not supported by filesystem root object.

5.3.2 Physical drive objects

5.3.2.1 Description

This message is used to close an opened disk/partition/file.

The user process sends an **SDD_DEV_CLOSE** request message to the controller process of a disk/partition/file object. The controller process replies with an **SDD_DEV_CLOSE_REPLY** reply message.

5.3.2.2 Message IDs

Request message	**SDD_DEV_CLOSE**
Reply message	**SDD_DEV_CLOSE_REPLY**

5.3.2.3 sdd_devClose_t Structure

```
typedef struct sdd_devClose_s {
  sdd_baseMessage_t base;
} sdd_devClose_t;
```

5.3.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies an opened disk/partition/file object to be closed.

5.3.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSNODEVMAN	Device manager not found.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Drive or partition or file is not opened.
EIO	Error occurred in the low level disk I/O layer.

5.4 SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY

5.4.1 Filesystem root object

This message is not supported by filesystem root object.

5.4.2 Physical drive objects

5.4.2.1 Description

This message is used to get or set certain properties of a filesystem object.

The user process sends an **SDD_DEV_IOCTL** request message to the controller process of a disk or partiton object. The controller process replies with an **SDD_DEV_IOCTL_REPLY** reply message.

5.4.2.2 Message IDs

Request message	**SDD_DEV_IOCTL**
Reply message	**SDD_DEV_IOCTL_REPLY**

5.4.2.3 sdd_devIoctl_t Structure

```
typedef struct sdd_devIoctl_s {
  sdd_baseMessage_t base;
  unsigned int cmd;
  int ret;
  unsigned long outlineArg;
  unsigned char inlineArg[1];
} sdd_devIoctl_t;
```

5.4.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies a disk or partition object.
cmd	Object specific command.
	Specifies an ioctl command to be executed on the object. Refer to 6.36.4 for list of supported commands.
ret	Return value.
	This is filesystem specific and for FATFS and SAFE FATFS it is always 0.
outlineArg	Command specific argument.
	If equals NULL, **inlineArg** is used.
inlineArg	Command specific included argument.
	Argument if **outlineArg** is not used. The size is variable and the whole argument is included.

5.4.3 Errors

base.error	Error code.
EACCES	Access denied.
EEXIST	Directory or file already exists.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALCMD	Invalid ioctl command.
EFSINVALNAME	Name is invalid.
EFSISDIRECTORY	Object is a directory.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSMSGTOOSMALL	Message size is too small.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition is not opened for raw access.
EFSOPENEDRDONLY	File opened in read-only mode.
EFSREADONLYFILE	File has got read-only attribute enabled.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMANYCOLL	Too many name collisions.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EINVAL	Object is not a disk/partition.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Command is not supported in current configuration or for this object type.
ERANGE	Returned value exceeds supported range.

5.5 SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY

5.5.1 Filesystem root object

This message is not supported by filesystem root object.

5.5.2 Physical drive objects

5.5.2.1 Description

This message is used to open a disk/partition/file object for read, write or read/write.

The user process sends an **SDD_DEV_OPEN** request message to the controller process of a disk/partition/file object. The controller process replies with an **SDD_DEV_OPEN_REPLY** reply message.

5.5.2.2 Message IDs

Request message	**SDD_DEV_OPEN**
Reply message	**SDD_DEV_OPEN_REPLY**

5.5.2.3 sdd_devOpen_t Structure

```
typedef struct sdd_devOpen_s {
  sdd_baseMessage_t base;
  flags_t flags;
} sdd_devOpen_t;
```

5.5.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies a disk/partition/file object to be opened.
flags	Disk/partition/file open flags.
	Used by the **request message** and contains BSD conform flags.
O_RDONLY	Open disk/partition/file for read only.
O_WRONLY	Open disk/partition/file for write only.
O_RDWR	Open disk/partition/file for read and write.
O_TRUNC	Truncate file to length zero.
O_APPEND	Sets the read/write pointer to the end of the file. Every write operation is performed at the end of file.

O_RDONLY	Open file in read-only mode. Initial file pointer position is at the beginning of a file.
O_WRONLY	Open file in write-only mode. Initial file pointer position is at the beginning of a file.
O_RDWR	Open file in read-write mode. Initial file pointer position is at the beginning of a file.
O_WRONLY | O_APPEND	Open file in write-only mode. Initial file pointer position is at

	the end of file. Every write operation is performed to the end of file. File seek and resize operations are ignored and return error.
O_WRONLY | O_APPEND | O_TRUNC	File is truncated and opened in write-only mode. Every write operation is performed to the end of file. File seek and resize operations are ignored and return error.
O_WRONLY | O_TRUNC	File is truncated and opened in write-only mode.
O_RDWR | O_APPEND	Open file in read-write mode. Initial file pointer position is at the end of file. Every write operation is performed to the end of file and file pointer remains there after write. File seek and resize operations are possible, but any subsequent write will move file pointer position back to the end of file.
O_RDWR | O_APPEND | O_TRUNC	File gets truncated and is opened in read-write mode. Every write operation is performed to the end of file and file pointer remains there after write. File seek and resize operations are possible, but any subsequent write will move file pointer position back to the end of file.
O_RDWR | O_TRUNC	File is truncated and opened in read-write mode.

5.5.2.5 Errors

base.error	Error code.
EACCES	Access denied.
EBUSY	Disk or partition or file already opened.
EEXIST	Directory or file already exists.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALFLAGS	Open flags combination is invalid.
EFSINVALNAME	Name is invalid.
EFSISDIRECTORY	Object is a directory.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSREADONLYFILE	File has got read-only attribute enabled.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMANYCOLL	Too many name collisions.

EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EINVAL	Object is not a file.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.

5.6 SDD_DEV_READ / SDD_DEV_READ_REPLY

5.6.1 Filesystem root object

This message is not supported by filesystem root object.

5.6.2 Physical drive objects

5.6.2.1 Description

This message is used to read data from a disk/partition/file. It can only be used if the disk/partition/file was successfully opened for read.

The user process sends an **SDD_DEV_READ** request message to the reader process of a disk/partition/file object. The reader process replies with an **SDD_DEV_READ_REPLY** reply message. The reply contains the read data.

5.6.2.2 Message IDs

Request message	**SDD_DEV_READ**
Reply message	**SDD_DEV_READ_REPLY**

5.6.2.3 sdd_devRead_t Structure

```
typedef struct sdd_devRead_s {
  sdd_baseMessage_t base;
  ssize_t size;
  ssize_t curpos;
  uint8_t *outlineBuf;
  uint8_t inlineBuf[1];
} sdd_devRead_t;
```

5.6.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies an opened disk/partition/file object to read data from.
size	Number of bytes to read.
	In the **request message** contains a number of bytes to read from the disk/partition/file. In the **reply message** contains a number of bytes actually read.
curpos	Current disk/partition/file pointer position.
	In the **reply message** contains disk/partition/file position from the disk/partition/file beginning, after the read operation.
outlineBuf	Pointer to a referenced buffer to store data to.
<readptr>	Used by the **reply message** and can contain a pointer to the buffer to put the data to. Not recommended as pointers should not be used in messages. Rather use **inlineBuf**.
0	The member **inlineBuf** is used.

inlineBuf	In-message buffer to store data to.
	Buffer used by the **reply message** if **outlineBuf** is not used. The size is variable and all data will be put into this buffer. The size of allocated SDD_DEV_READ message must be big enough to fit the requested data size.

5.6.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSMSGTOOSMALL	Message size is too small for requested read size.
EFSNEGATIVESIZE	Read size cannot be negative.
EFSNODEVMAN	Device manager not found.
EFSNOTALIGNEDSIZE	Read size is not a multiple of sector size for disk/partition raw access.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition or file is not opened.
EFSOPENEDWRONLY	File opened in write-only mode.
EIO	Error occurred in the low level disk I/O layer.
ERANGE	Reading beyond drive or partition boundaries.

5.7 SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY

5.7.1 Filesystem root object

This message is not supported by filesystem root object.

5.7.2 Physical drive objects

5.7.2.1 Description

This message is used to write data to a disk/partition/file. It can only be used if the disk/partition/file was successfully opened for write.

The user process sends an **SDD_DEV_WRITE** request message to the writer process of a disk/partition/file object. The writer process replies with an **SDD_DEV_WRITE_REPLY** reply message. The reply contains the number of bytes written.

5.7.2.2 Message IDs

Request message	**SDD_DEV_WRITE**
Reply message	**SDD_DEV_WRITE_REPLY**

5.7.2.3 sdd_devWrite_t Structure

```
typedef struct sdd_devWrite_s {
  sdd_baseMessage_t base;
  ssize_t size;
  ssize_t curpos;
  const __u8 *outlineBuf;
  __u8 inlineBuf[1];
} sdd_devWrite_t;
```

5.7.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies an opened disk/partition/file object to write data to.
size	Number of bytes to write.
	In the **request message** contains a number of bytes to write to the disk/partition/file. In the **reply message** contains a number of bytes actually written.
curpos	Current disk/partition/file pointer position.
	In the **reply message** contains disk/partition/file position from the disk/partition/file beginning, after the write operation.
outlineBuf	Pointer to a referenced buffer to get the data from.
<readptr>	Used by the **request message** and can contain a pointer to the buffer to get the data for writing to the disk/partition/file. Not recommended as pointers should not be used in messages. Rather use **inlineBuf**. Not used by the **reply message** and can have any value.

0	The member **inlineBuf** is used.

inlineBuf	In-message buffer containing data to write.
	Buffer used by the **request message** if **outlineBuf** is not used. The size is variable and all data for writing will be taken from this buffer. The allocated SDD_DEV_WRITE message must be big enough to contain all the data that are requested to be written. Not used by the **reply message** and can have any value.

5.7.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSMSGTOOSMALL	Message size is too small for requested write size.
EFSNEGATIVESIZE	Write size cannot be negative.
EFSNODEVMAN	Device manager not found.
EFSNOTALIGNEDSIZE	Write size is not a multiple of sector size for disk/partition raw access.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition or file is not opened.
EFSOPENEDRDONLY	File opened in read-only mode.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Error occurred in the low level disk I/O layer.
ERANGE	Reading beyond drive or partition boundaries.

5.8 SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY

5.8.1 Filesystem root object

This message is not supported by filesystem root object.

5.8.2 Physical drive objects

5.8.2.1 Description

This message is used to resize an opened file.

The user process sends an **SDD_FILE_RESIZE** request message to the controller process of a file object. The controller process replies with an **SDD_FILE_RESIZE_REPLY** reply message.

Refer to 6.23 for additional rules when using **SDD_FILE_RESIZE** message.

5.8.2.2 Message IDs

Request message	**SDD_FILE_RESIZE**
Reply message	**SDD_FILE_RESIZE_REPLY**

5.8.2.3 sdd_fileResize_t Structure

```
typedef struct sdd_fileResize_s {
  sdd_baseMessage_t base;
  ssize_t size;
} sdd_fileResize_t;
```

5.8.2.4 Structure Members

base	File object descriptor.
	Specifies an opened file object to resize.
size	New file size in range 0..2GB-1
	In the **request message** contains a new size of a file.
	In the **reply message** contains an actual size of a file after resizing.

5.8.2.5 Errors

base.error	Error code.
EFSAPPENDONLY	File is opened for appending only.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSISDRIVE	Object is a drive, not file.
EFSISPARTITION	Object is a partition, not file.

EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNEGATIVESIZE	New file size cannot be negative.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSOPENEDRDONLY	File opened in read-only mode.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Error occurred in the low level disk I/O layer.

5.9 SDD_FILE_RESIZE64 / SDD_FILE_RESIZE64_REPLY

5.9.1 Filesystem root object

This message is not supported by filesystem root object.

5.9.2 Physical drive objects

5.9.2.1 Description

This message is used to resize an opened file.

The user process sends an **SDD_FILE_RESIZE64** request message to the controller process of a file object. The controller process replies with an **SDD_FILE_RESIZE64_REPLY** reply message.

Refer to 6.24 for additional rules when using **SDD_FILE_RESIZE64** message.

5.9.2.2 Message IDs

Request message	**SDD_FILE_RESIZE64**
Reply message	**SDD_FILE_RESIZE64_REPLY**

5.9.2.3 sdd_fileResize64_t Structure

```
typedef struct sdd_fileResize64_s {
  sdd_baseMessage_t base;
  uint64_t size;
} sdd_fileResize64_t;
```

5.9.2.4 Structure Members

base	File object descriptor.
	Specifies an opened file object to resize.
size	New file size in range 0..4GB-1
	In the **request message** contains a new size of a file.
	In the **reply message** contains an actual size of a file after resizing.

5.9.2.5 Errors

base.error	Error code.
EFSAPPENDONLY	File is opened for appending only.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSISDRIVE	Object is a drive, not file.
EFSISPARTITION	Object is a partition, not file.

EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSOPENEDRDONLY	File opened in read-only mode.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Error occurred in the low level disk I/O layer.
ERANGE	Requested file size is too big (> 4GB-1).

5.10 SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY

5.10.1 Filesystem root object

This message is not supported by filesystem root object.

5.10.2 Physical drive objects

5.10.2.1 Description

This message is used to change current read/write offset of an opened disk/partition/file.

The user process sends an **SDD_FILE_SEEK** request message to the controller process of a disk/partition/file object. The controller process replies with an **SDD_FILE_SEEK_REPLY** reply message.

Refer to 6.25.1 for additional rules when using **SDD_FILE_SEEK** message.

5.10.2.2 Message IDs

Request message	**SDD_FILE_SEEK**
Reply message	**SDD_FILE_SEEK_REPLY**

5.10.2.3 sdd_fileSeek_t Structure

```
typedef struct sdd_fileSeek_s {
  sdd_baseMessage_t base;
  off_t offset;
  int whence;
} sdd_fileSeek_t;
```

5.10.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies an opened disk/partition/file object to change current position of.
offset	New disk/partition/file offset.
	In the **request message** contains a new disk/partition/file read/write offset.
	In the **reply message** contains an actual offset from the beginning of a disk/partition/file after changing current disk/partition/file position.
whence	Offset origin.
SEEK_CUR	**offset** is relative to the current position.
SEEK_END	**offset** is relative to the end of the disk/partition/file.
SEEK_SET	**offset** is relative to the beginning of the disk/partition/file.

5.10.2.5 Errors

base.error	Error code.
EBIG	Requested position exceeds limit of seek function, use 64-bit version.

EFSAPPENDONLY	File is opened for appending only.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition or file is not opened.
EINVAL	Offset is not a multiple of sector size for disk/partition raw access or specified origin is invalid.
EIO	Error occurred in the low level disk I/O layer.
ERANGE	Requested position goes beyond beginning or end of disk/partition/file.

5.11 SDD_FILE_SEEK64 / SDD_FILE_SEEK64_REPLY

5.11.1 Filesystem root object

This message is not supported by filesystem root object.

5.11.2 Physical drive objects

5.11.2.1 Description

This message is used to change current read/write offset of an opened disk, partition or file.

The user process sends an **SDD_FILE_SEEK64** request message to the controller process of a disk, partition or file. The controller process replies with an **SDD_FILE_SEEK64_REPLY** reply message.

Refer to 6.26.1 for additional rules when using **SDD_FILE_SEEK64** message.

5.11.2.2 Message IDs

Request message	**SDD_FILE_SEEK64**
Reply message	**SDD_FILE_SEEK64_REPLY**

5.11.2.3 sdd_fileSeek64_t Structure

```
typedef struct sdd_fileSeek64_s {
  sdd_baseMessage_t base;
  int64_t offset;
  int whence;
} sdd_fileSeek64_t;
```

5.11.2.4 Structure Members

base	Disk, partition or file object descriptor.
	Specifies an opened disk/partition/file object to change current position of.
offset	New disk/partition/file offset.
	In the **request message** contains a new disk/partition/file read/write offset. In the **reply message** contains an actual offset from the beginning of a disk/partition/file after changing current disk/partition/file position.
whence	Offset origin.
SEEK_CUR	**offset** is relative to the current position.
SEEK_END	**offset** is relative to the end of the disk/partition/file.
SEEK_SET	**offset** is relative to the beginning of the disk/partition/file.

5.11.2.5 Errors

base.error	Error code.
EFSAPPENDONLY	File is opened for appending only.

EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition or file is not opened.
EINVAL	Offset is not a multiple of sector size for disk/partition raw access or specified origin is invalid.
EIO	Error occurred in the low level disk I/O layer.
ERANGE	Requested position goes beyond beginning or end of disk/partition/file.

5.12 SDD_MAN_ADD / SDD_MAN_ADD_REPLY

5.12.1 Message IDs

Request message	**SDD_MAN_ADD**
Reply message	**SDD_MAN_ADD_REPLY**

5.12.2 sdd_manAdd_t Structure

```
typedef struct sdd_manAdd_s {
  sdd_obj_t object;
} sdd_manAdd_t;
```

5.12.3 Filesystem root object

This message is not supported by filesystem root object.

5.12.4 Physical drive objects

5.12.4.1 Description

This message is used to add a new directory or a new file to an existing directory.

The user process sends an **SDD_MAN_ADD** request message to the controller process of a directory object. The message includes an existing directory handle and a name and a type of a new object. The controller creates a new directory or a new file on a filesystem partition. The controller sends **SDD_MAN_ADD_REPLY** reply message back.

5.12.4.2 Structure Members

object.manager	Base manager (directory) object descriptor.
	Specifies a handle of a manager (directory) to which a new object shall be added.

object.name	New object name.

object.type	New object type.
SFS_ATTR_DIR	A new object type is a directory.
SFS_ATTR_FILE	A new object type is a file.

5.12.4.3 Errors

base.error	Error code.
EACCES	Access denied.
EEXIST	Directory or file already exists.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.

EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSEMPTYNAME	New item name cannot be empty.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	New item name is corrupted. Alternatively: Name is invalid.
EFSINVALTYPE	Requested new item type is invalid.
EFSISDIRECTORY	Object is a directory.
EFSISDRIVE	Cannot add a new item to drive root.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSREADONLYFILE	File has got read-only attribute enabled.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMANYCOLL	Too many name collisions.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.

5.13 SDD_MAN_GET / SDD_MAN_GET_REPLY

5.13.1 Message IDs

Request message	**SDD_MAN_GET**
Reply message	**SDD_MAN_GET_REPLY**

5.13.2 sdd_manGet_t Structure

```
typedef struct sdd_manGet_s {
  sdd_obj_t object;
} sdd_manGet_t;
```

5.13.3 Filesystem root object

5.13.3.1 Description

This message is used to get a physical drive object descriptor (including the process IDs and handle).

The user process sends an **SDD_MAN_GET** request message to the controller process of a filesystem root object. The controller process replies with an **SDD_MAN_GET_REPLY** reply message. The reply message contains all informations about physical drive (including all process IDs).

5.13.3.2 Structure Members

object.base	Physical drive object descriptor. In **reply message** this field contains requested physical drive object descriptor.
object.name	Physical drive name. This field contains a name of a physical drive to get.
object.manager	Not used and must be NULL.

5.13.3.3 Errors

base.error	Error code.
EFSEMPTYNAME	Drive name to get is empty.
EFSINVALNAME	Drive name to get is corrupted.
EFSNOTROOT	Object is not a filesystem root.
ENOENT	There is no drive with specified name to get or there is no upper level item to get.
ENOMEM	Out of memory.

5.13.4 Physical drive objects

5.13.4.1 Description

This message is used to get a partition, a directory or a file object descriptor (including the process IDs and handle).

The user process sends an **SDD_MAN_GET** request message to the controller process of:

- a physical drive, when getting a partition object
- a partition, when getting a directory/file from the first level of the partition tree.
- a directory, when getting a directory/file from the level lower than partition tree root.

The controller process replies with an **SDD_MAN_GET_REPLY** reply message. The reply message contains all informations about requested object (including all process IDs).

5.13.4.2 Structure Members

object.base	Requested object descriptor (reply message).
	In **reply message** this field contains requested partition, directory or file object descriptor.

object.name	Requested object name (request and reply message).
	This field contains a name of a partition, a directory or a file to get.

object.manager	Handle of a manager of the requested object (request message).
NULL	In **request message** if requested object is a partition then this value must be NULL.
!= NULL	In **request message** if requested object is a directory or a file then this value is a handle of a partition or an upper directory.

5.13.4.3 Errors

base.error	Error code.
EBUSY	Cannot grant raw access to mounted partition or partition is already being accessed in raw mode.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSEMPTYNAME	Item name cannot be empty.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
EFSRODRIVE	The physical drive is write protected.

EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOENT	Specified partition is not mounted.
ENOMEM	Out of memory.

5.14 SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY

5.14.1 Message IDs

Request message	**SDD_MAN_GET_FIRST**
Reply message	**SDD_MAN_GET_FIRST_REPLY**

5.14.2 sdd_manGetFirst_t Structure

```
typedef struct sdd_manGetFirst_s {
  sdd_obj_t object;
} sdd_manGetFirst_t;
```

5.14.3 Filesystem root object

5.14.3.1 Description

This message is used to get first physical drive object descriptor of all physical drives added to the filesystem (including the process IDs and handle).

The user process sends an **SDD_MAN_GET_FIRST** request message to the controller process of a filesystem root object. The controller process replies with an **SDD_MAN_GET_FIRST_REPLY** reply message. The reply message contains all informations about physical drive (including all process IDs).

5.14.3.2 Structure Members

object.base	Physical drive object descriptor (reply message). In **reply message** this field contains first physical drive object descriptor.
object.manager	Not used and must be NULL.

5.14.3.3 Errors

base.error	Error code.
EFSNOTROOT	Object is not a filesystem root.
ENOENT	No drives found.
ENOMEM	Out of memory.

5.14.4 Physical drive objects

5.14.4.1 Description

This message is used to get first mounted partition on a physical drive or first directory/file object descriptor in the specified directory (including the process IDs and handle).

The user process sends an **SDD_MAN_GET_FIRST** request message to the controller process of:

- a physical drive, when getting first mounted partition object
- a partition, when getting first directory/file from the first level of the partition tree.

- a directory, when getting first directory/file from the level lower than partition tree root.

The controller process replies with an **SDD_MAN_GET_FIRST_REPLY** reply message. The reply message contains all informations about requested object (including all process IDs).

5.14.4.2 Structure Members

object.base	First object descriptor (reply message).
	In **reply message** this field contains first mounted requested partition or first directory/file object descriptor.

object.manager	Handle of a manager of the requested object (request message).
NULL	In **request message** if requested first object is a partition then this value must be NULL.
!= NULL	In **request message** if requested first object is a directory/file then this value is a handle of a partition or an upper directory.

5.14.4.3 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSISFILE	Object is a file, not directory.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	Name is too long to fit into message. Alternatively: File or path name too long.
ENOENT	No partition found on a drive or no items in a directory.
ENOMEM	Out of memory.

5.15 SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY

5.15.1 Message IDs

Request message	**SDD_MAN_GET_NEXT**
Reply message	**SDD_MAN_GET_NEXT_REPLY**

5.15.2 sdd_manGetNext_t Structure

```
typedef struct sdd_manGetNext_s {
  sdd_obj_t object;
} sdd_manGetNext_t;
```

5.15.3 Filesystem root object

5.15.3.1 Description

This message is used to get next physical drive object descriptor (including the process IDs and handle) after the one specified in a request message.

The user process sends an **SDD_MAN_GET_NEXT** request message to the controller process of a filesystem root object. The controller process replies with an **SDD_MAN_GET_NEXT_REPLY** reply message. The reply message contains all informations about physical drive (including all process IDs).

5.15.3.2 Structure Members

object.base	Physical drive object descriptor (reply message). In **reply message** this field contains next physical drive object descriptor.
object.name	Previous object name (request message) or next object name (reply message). In **request message** this field contains the name of the previous physical drive. In **reply message** this field contains the name of a next physical drive.
object.manager	Not used and must be NULL.

5.15.3.3 Errors

base.error	Error code.
EFSEMPTYNAME	Previous drive name cannot be empty.
EFSINVALNAME	Previous drive name is corrupted.
EFSNOTROOT	Object is not a filesystem root.
ENOENT	Previous drive not found or no more drives.
ENOMEM	Out of memory.

5.15.4 Physical drive objects

5.15.4.1 Description

This message is used to get next mounted partition on a physical drive or next directory/file object descriptor in the specified directory (including the process IDs and handle).

The user process sends an **SDD_MAN_GET_NEXT** request message to the controller process of:

- a physical drive, when getting next mounted partition object
- a partition, when getting next directory/file from the first level of the partition tree.
- a directory, when getting next directory/file from the level lower than partition tree root.

The controller process replies with an **SDD_MAN_GET_NEXT_REPLY** reply message. The reply message contains all informations about requested object (including all process IDs).

5.15.4.2 Structure Members

object.base	Next object descriptor (reply message).
	In **reply message** this field contains next mounted requested partition or next directory/file object descriptor.

object.name	Previous object name (request message) or next object name (reply message).
	In **request message** this field contains the name of the previous partition, directory or file. In **reply message** this field contains the name of a next partition, directory or file.

object.manager	Handle of a manager of the requested object (request message).
NULL	In **request message** if requested next object is a partition then this value must be NULL.
!= NULL	In **request message** if requested next object is a directory/file then this value is a handle of a partition or an upper directory.

5.15.4.3 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSEMPTYNAME	Previous object name cannot be empty.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Previous object name is corrupted or previous partition name is invalid. Alternatively: Name is invalid.
EFSISFILE	Object is a file, not directory.
EFSISRAWOBJECT	Object is disk or partition for raw access.

EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	Name is too long to fit into message. Alternatively: File or path name too long.
ENOENT	Previous partition is not mounted anymore or no more partitions found on a drive or no more items found in a directory.
ENOMEM	Out of memory.

5.16 SDD_MAN_NOTIFY_ADD / SDD_MAN_NOTIFY_ADD_REPLY

5.16.1 Message IDs

Request message	**SDD_MAN_NOTIFY_ADD**
Reply message	**SDD_MAN_NOTIFY_ADD_REPLY**

5.16.2 sdd_manNotify_t Structure

```
typedef struct sdd_manNotify_s {
  sdd_name_t name;
  sc_ticks_t tmo;
  sc_tmoid_t tmoid;
} sdd_manNotify_t;
```

5.16.3 Filesystem root object

5.16.3.1 Description

This message is used to request a message from the filesystem root object when a physical drive is added to the filesystem tree.

The user process sends an **SDD_MAN_NOTIFY_ADD** request message to the controller process of a root filesystem object. The controller process replies with an **SDD_MAN_NOTIFY_ADD_REPLY** reply message when a physical drive has been added.

5.16.3.2 Structure Members

name.base	Root filesystem object descriptor.
	Specifies a root filesystem object descriptor which shall notify about physical drive add.
name.name	Name of the physical drive.
	Used in the request message to specify the name of a physical drive to notify about.
tmo	Time-out.
	If a physical drive is not added within this time-out, filesystem root object will reply with **SDD_MAN_NOTIFY_ADD_REPLY** message containing an error code.
tmoid	Internal use.
	Used by the filesystem for its own purposes.

5.16.3.3 Errors

base.error	Error code.
EFSDRVPROCEXITED	Drive process exited abnormally.
EFSEMPTYNAME	Drive name is empty.
EFSINVALNAME	Drive name is invalid.

ENOMEM	No memory to add new notification.
ETIMEDOUT	Timeout waiting for drive add.

5.16.4 Physical drive objects

5.16.4.1 Description

This message is used to request a message from the physical drive object when a partition is mounted.

The user process sends an **SDD_MAN_NOTIFY_ADD** request message to the controller process of a physical drive object. The controller process replies with an **SDD_MAN_NOTIFY_ADD_REPLY** reply message when a partition has been mounted.

5.16.4.2 Structure Members

name.base	Physical drive object descriptor.
	Specifies a physical drive object descriptor which shall notify about partition mount.
name.name	Name of the partition.
	Used in the request message to specify the name of a partition to notify about.
tmo	Time-out.
	If a partition is not mounted within this time-out, filesystem root object will reply with **SDD_MAN_NOTIFY_ADD_REPLY** message containing an error code.
tmoid	Internal use.
	Used by the filesystem for its own purposes.

5.16.4.3 Errors

base.error	Error code.
EFSDRIVEREMOVED	Drive has been removed.
EFSINVALNAME	Specified partition name is invalid.
EFSNOTPARTITION	Cannot notify about adding directory/file, only drive or partition.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
ENOMEM	No memory to add new notification.
ENOTSUP	Notification for raw access is not supported.
ETIMEDOUT	Timeout waiting for partition add.

5.17 SDD_MAN_NOTIFY_RM / SDD_MAN_NOTIFY_RM_REPLY

5.17.1 Message IDs

Request message	**SDD_MAN_NOTIFY_RM**
Reply message	**SDD_MAN_NOTIFY_RM_REPLY**

5.17.2 sdd_manNotify_t Structure

```
typedef struct sdd_manNotify_s {
  sdd_name_t name;
  sc_ticks_t tmo;
  sc_tmoid_t tmoid;
} sdd_manNotify_t;
```

5.17.3 Filesystem root object

5.17.3.1 Description

This message is used to request a message from the filesystem root object when a physical drive is removed from the filesystem tree.

The user process sends an **SDD_MAN_NOTIFY_RM** request message to the controller process of a root filesystem object. The controller process replies with an **SDD_MAN_NOTIFY_RM_REPLY** reply message when a physical drive has been removed.

5.17.3.2 Structure Members

name.base	Root filesystem object descriptor.
	Specifies a root filesystem object descriptor which shall notify when physical drive has been removed.
name.name	Name of the physical drive.
	Used in the request message to specify the name of a physical drive to notify about.
tmo	Time-out.
	If a physical drive is not removed within this time-out, filesystem root object will reply with **SDD_MAN_NOTIFY_RM_REPLY** message containing an error code.
tmoid	Internal use.
	Used by the filesystem for its own purposes.

5.17.3.3 Errors

base.error	Error code.
EFSEMPTYNAME	Drive name is empty.
EFSINVALNAME	Drive name is invalid.
ENOMEM	No memory to add new notification.

ETIMEDOUT	Timeout waiting for drive remove.

5.17.4 Physical drive objects

5.17.4.1 Description

This message is used to request a message from the physical drive object when a partition is unmounted.

The user process sends an **SDD_MAN_NOTIFY_RM** request message to the controller process of a physical drive object. The controller process replies with an **SDD_MAN_NOTIFY_RM_REPLY** reply message when a partition has been unmounted.

5.17.4.2 Structure Members

name.base	Physical drive object descriptor.
	Specifies a physical drive object descriptor which shall notify about partition unmount.
name.name	Name of the partition.
	Used in the request message to specify the name of a partition to notify about.
tmo	Time-out.
	If a partition is not unmounted within this time-out, filesystem root object will reply with **SDD_MAN_NOTIFY_RM_REPLY** message containing an error code.
tmoid	Internal use.
	Used by the filesystem for its own purposes.

5.17.4.3 Errors

base.error	Error code.
EFSDRIVEREMOVED	Drive has been removed.
EFSINVALNAME	Specified partition name is invalid.
EFSNOTPARTITION	Cannot notify about removing directory/file, only drive or partition.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
ENOMEM	No memory to add new notification.
ENOTSUP	Notification for raw access is not supported.
ETIMEDOUT	Timeout waiting for partition remove.

5.18 SDD_MAN_RM / SDD_MAN_RM_REPLY

5.18.1 Message IDs

Request message	**SDD_MAN_RM**
Reply message	**SDD_MAN_RM_REPLY**

5.18.2 sdd_manRm_t Structure

```
typedef struct sdd_manRm_s {
  sdd_obj_t object;
} sdd_manRm_t;
```

5.18.3 Filesystem root object

This message is not supported by filesystem root object.

5.18.4 Physical drive objects

5.18.4.1 Description

This message is used to remove a directory or a file from an existing directory.

The user process sends an **SDD_MAN_RM** request message to the controller process of a directory object. The message includes a descriptor of a directory/file to remove. The controller sends **SDD_MAN_RM_REPLY** reply message back.

5.18.4.2 Structure Members

object.manager	Base manager (directory) object descriptor.
	Specifies a handle of a manager (directory) from which an object shall be removed.

object.name	Name of an object which shall be removed.

5.18.4.3 Errors

base.error	Error code.
EACCES	Access denied.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSEMPTYNAME	Item name for removing cannot be empty.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Item name for removing is corrupted. Alternatively: Name is invalid.

EFSISDRIVE	Cannot remove an item from drive root.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTEMPTYDIR	Directory is not empty.
EFSNOTFSOBJECT	Not filesystem object.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSREADONLYDIR	Directory has got read-only attribute enabled.
EFSREADONLYFILE	File has got read-only attribute enabled.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.

5.19 SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY

5.19.1 Message IDs

Request message	**SDD_OBJ_DUP**
Reply message	**SDD_OBJ_DUP_REPLY**

5.19.2 sdd_objDup_t Structure

```
typedef struct sdd_objDup_s {
  sdd_baseMessage_t base;
} sdd_objDup_t;
```

5.19.3 Filesystem root object

5.19.3.1 Description

This message is used to create a copy of an object representing a filesystem root.

The user process sends an **SDD_OBJ_DUP** request message to the controller process of an object to be duplicated. The controller sends **SDD_OBJ_DUP_REPLY** reply message back.

5.19.3.2 Structure Members

base	Object descriptor.
	In **request message** this field contains a descriptor of a filesystem root object to be duplicated. In **reply message** this field contains a descriptor of a copy of a filesystem root object.

5.19.3.3 Errors

base.error	Error code.
EFSNOTROOT	Object is not a filesystem root.

5.19.4 Physical drive objects

5.19.4.1 Description

This message is used to create a copy of an object representing a physical drive, partition, directory or file.

The user process sends an **SDD_OBJ_DUP** request message to the controller process of an object to be duplicated. The controller sends **SDD_OBJ_DUP_REPLY** reply message back.

5.19.4.2 Structure Members

base	Object descriptor.
	In **request message** this field contains a descriptor of an object to be duplicated. In **reply message** this field contains a descriptor of a copy of an object.

5.19.4.3 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOTFSOBJECT	Not filesystem object.
ENOMEM	Out of memory.

5.20 SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY

5.20.1 Message IDs

Request message	**SDD_OBJ_RELEASE**
Reply message	**SDD_OBJ_RELEASE_REPLY**

5.20.2 sdd_objRelease_t Structure

```
typedef struct sdd_objRelease_s {
  sdd_baseMessage_t base;
} sdd_objRelease_t;
```

5.20.3 Filesystem root object

5.20.3.1 Description

This message is used to release a temporary object representing a filesystem root.

The user process sends an **SDD_OBJ_RELEASE** request message to the controller process of an object to be released. The controller sends **SDD_OBJ_RELEASE_REPLY** reply message back.

5.20.3.2 Structure Members

base	Object descriptor.
	In **request message** this field contains a descriptor of a temporary filesystem root object to be released.

5.20.3.3 Errors

No errors returned.

5.20.4 Physical drive objects

5.20.4.1 Description

This message is used to release a temporary object representing a physical drive, partition, directory or file.

The user process sends an **SDD_OBJ_RELEASE** request message to the controller process of an object to be released. The controller sends **SDD_OBJ_RELEASE_REPLY** reply message back.

5.20.4.2 Structure Members

base	Object descriptor.
	In **request message** this field contains a descriptor of a temporary object to be released.

5.20.4.3 Errors

base.error	Error code.

EBUSY	Cannot release object which is opened.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTREFERENCED	Releasing not referenced object.

5.21 SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY

5.21.1 Filesystem root object

This message is not supported by filesystem root object.

5.21.2 Physical drive objects

5.21.2.1 Description

This message is used get a size of a filesystem object: physical drive, partition, directory or file.

The user process sends an **SDD_OBJ_SIZE_GET** request message to the reader process of a file object. The controller process replies with an **SDD_OBJ_SIZE_GET_REPLY** reply message. The reply contains the calculated sizes.

Maximum size returned cannot exceed 4GB-1. To get larger sizes use **SDD_OBJ_SIZE64_GET** message (section 5.22).

5.21.2.2 Message IDs

Request message	**SDD_OBJ_SIZE_GET**
Reply message	**SDD_OBJ_SIZE_GET_REPLY**

5.21.2.3 sdd_objSize_t Structure

```
typedef struct sdd_objSize_s {
  sdd_baseMessage_t base;
  size_t total;
  size_t free;
  size_t bad;
} sdd_objSize_t;
```

5.21.2.4 Structure Members

base	Object descriptor.
	Specifies a filesystem to object to calculate the sizes of.
total	Total size of a filesystem object.
free	Free available size of a filesystem object.
bad	Not available size of a filesystem object.

5.21.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.

EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EINVAL	Partition number is invalid.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	This operation is not supported in read-only configuration.
ERANGE	Calculated size exceeds limit the can be returned by this operation (4GB-1).

5.22 SDD_OBJ_SIZE64_GET / SDD_OBJ_SIZE64_GET_REPLY

5.22.1 Filesystem root object

This message is not supported by filesystem root object.

5.22.2 Physical drive objects

5.22.2.1 Description

This message is used get a size of a filesystem object: physical drive, partition, directory or file.

The user process sends an **SDD_OBJ_SIZE64_GET** request message to the reader process of a file object. The controller process replies with an **SDD_OBJ_SIZE64_GET_REPLY** reply message. The reply contains the calculated sizes.

5.22.2.2 Message IDs

Request message	**SDD_OBJ_SIZE64_GET**
Reply message	**SDD_OBJ_SIZE64_GET_REPLY**

5.22.2.3 sdd_objSize64_t Structure

```
typedef struct sdd_objSize64_s {
  sdd_baseMessage_t base;
  uint64_t total;
  uint64_t free;
  uint64_t bad;
} sdd_objSize64_t;
```

5.22.2.4 Structure Members

base	Object descriptor. Specifies a filesystem to object to calculate the sizes of.
total	Total size of a filesystem object.
free	Free available size of a filesystem object.
bad	Not available size of a filesystem object.

5.22.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.

EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EINVAL	Partition number is invalid.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	This operation is not supported in read-only configuration.

5.23 SDD_OBJ_TAG_GET / SDD_OBJ_TAG_GET_REPLY

5.23.1 Filesystem root object

This message is not supported by filesystem root object.

5.23.2 Physical drive objects

5.23.2.1 Description

This message is used to get a specific property of a filesystem object.

The user process sends an **SDD_OBJ_TAG_GET** request message to the controller process of a filesystem object. The controller process replies with an **SDD_OBJ_TAG_GET_REPLY** reply message. The reply contains the value of a requested property.

5.23.2.2 Message IDs

Request message	**SDD_OBJ_TAG_GET**
Reply message	**SDD_OBJ_TAG_GET_REPLY**

5.23.2.3 sdd_objIoctl_t Structure

```
typedef struct sdd_objIoctl_s {
  sdd_baseMessage_t base;
  union {
    sc_tab_t entity[1];
    sc_ioctl_t ioctl;
  } u;
} sdd_objIoctl_t;
```

5.23.2.4 Structure Members

base	Object descriptor.
	Specifies a filesystem object to get a property of.
u.ioctl.tag	Property type to get.
	In a **request message** this field specifies a property type to get.
u.ioctl.data	Property data.
	In a **reply message** this field contains a property value.
u.entiry	This field is not used.

5.23.2.5 Supported properties types

Refer to 6.34.4 for a complete list of supported properties types.

5.23.2.6 Errors

base.error	Error code.

EACCES	Access denied.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSINVALPROPERTY	Specified property tag is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSMSGTOOSMALL	Message size is too small.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSRODRIVE	The physical drive is write protected.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Cannot get file/directory label/serial number or cannot get attributes of a drive/partition or cannot get file cache size of a drive/partition or cannot get FAT cache size of directory/file or cannot get allocated objects list for directory/file.

5.24 SDD_OBJ_TAG_SET / SDD_OBJ_TAG_SET_REPLY

5.24.1 Filesystem root object

This message is not supported by filesystem root object.

5.24.2 Physical drive objects

5.24.2.1 Description

This message is used to set a specific property of a filesystem object.

The user process sends an **SDD_OBJ_TAG_SET** request message to the controller process of a filesystem object. The controller process replies with an **SDD_OBJ_TAG_SET_REPLY** reply message.

5.24.2.2 Message IDs

Request message	**SDD_OBJ_TAG_SET**
Reply message	**SDD_OBJ_TAG_SET_REPLY**

5.24.2.3 sdd_objIoctl_t Structure

```
typedef struct sdd_objIoctl_s {
  sdd_baseMessage_t base;
  union {
    sc_tab_t entity[1];
    sc_ioctl_t ioctl;
  } u;
} sdd_objIoctl_t;
```

5.24.2.4 Structure Members

base	Object descriptor.
	Specifies a filesystem object to set a property of.
u.ioctl.tag	Property type to set.
	In a **request message** this field specifies a property type to set.
u.ioctl.data	Property data.
	In a **request message** this field contains a new property value.
u.entiry	This field is not used.

5.24.2.5 Supported properties types

Refer to 6.42.4 for a complete list of supported properties types.

5.24.2.6 Errors

base.error	Error code.

EACCES	Access denied.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALLABEL	Partition label is invalid.
EFSINVALNAME	Name is invalid.
EFSINVALPROPERTY	Specified property tag is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSRODRIVE	The physical drive is write protected.
EFSZEROSIZE	File cache size cannot be zero.
EINVAL	Message size is invalid.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Cannot set file/directory label/serial number or cannot set attributes of a drive/partition or cannot set file cache size of a drive/partition or cannot set FAT cache size of file/directory.

5.25 SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY

5.25.1 Filesystem root object

This message is not supported by filesystem root object.

5.25.2 Physical drive objects

5.25.2.1 Description

This message is used to get the last modification time of a directory/file.

The user process sends an **SDD_OBJ_TIME_GET** request message to the controller process of a file object. The controller process replies with an **SDD_OBJ_TIME_GET_REPLY** reply message. The reply contains the modification time.

5.25.2.2 Message IDs

Request message	**SDD_OBJ_TIME_GET**
Reply message	**SDD_OBJ_TIME_GET_REPLY**

5.25.2.3 sdd_objTime_t Structure

```
typedef struct sdd_objTime_s {
  sdd_baseMessage_t base;
  __u32 data;
} sdd_objTime_t;
```

5.25.2.4 Structure Members

base	Object descriptor.
	Specifies a filesystem object to get a modification time of.
data	Modification time.

5.25.2.5 Time decoding

For FAT filesystem the returned time is encoded using following format:

bits <31:25>	Year offset from 1980 (0..127)
bits <24:21>	Month (1..12)
bits <20:16>	Day (1..31)
bits <15:11>	Hour (0..23)
bits <10:5>	Minute (0..59)
bits <4:0>	Second / 2 (0..29)

A set of macros is provided for decoding each field of returned **data**:

`CHANFS_OBJECTTIME_EXTRACT_YEAR(data)` – Extract year.

`CHANFS_OBJECTTIME_EXTRACT_MONTH(data)` – Extract month.

`CHANFS_OBJECTTIME_EXTRACT_DAY(data)` – Extract day.

`CHANFS_OBJECTTIME_EXTRACT_HOUR(data)` – Extract hour.

`CHANFS_OBJECTTIME_EXTRACT_MINUTE(data)` – Extract minute.

`CHANFS_OBJECTTIME_EXTRACT_SECOND(data)` – Extract second.

5.25.2.6 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EINVAL	Cannot get time of a drive/partition.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Cannot get time of a drive/partition.

5.26 SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY

5.26.1 Filesystem root object

This message is not supported by filesystem root object.

5.26.2 Physical drive objects

5.26.2.1 Description

This message is used to set the last modification time of a directory/file.

The user process sends an **SDD_OBJ_TIME_SET** request message to the controller process of a file object. The controller process replies with an **SDD_OBJ_TIME_SET_REPLY** reply message.

5.26.2.2 Message IDs

Request message	**SDD_OBJ_TIME_SET**
Reply message	**SDD_OBJ_TIME_SET_REPLY**

5.26.2.3 sdd_objTime_t Structure

```
typedef struct sdd_objTime_s {
  sdd_baseMessage_t base;
  __u32 data;
} sdd_objTime_t;
```

5.26.2.4 Structure Members

base	Object descriptor.
	Specifies a filesystem object to set a modification time of.
data	Modification time.

5.26.2.5 Time encoding

For FAT filesystem the time has to be encoded in the following format:

bits <31:25>	Year offset from 1980 (0..127)
bits <24:21>	Month (1..12)
bits <20:16>	Day (1..31)
bits <15:11>	Hour (0..23)
bits <10:5>	Minute (0..59)
bits <4:0>	Second / 2 (0..29)

A macro is provided to encode the date/time to FAT filesystem format:

```
CHANFS_OBJECTTIME_CONVERT(year, month, day, hour, minute, second)
```

5.26.2.6 Errors

base.error	Error code.
EACCES	Access denied.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSRODRIVE	The physical drive is write protected.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Cannot set time of a drive/partition.

5.27 SFS_ASSIGN / SFS_ASSIGN_REPLY

5.27.1 Message IDs

Request message	**SFS_ASSIGN**
Reply message	**SFS_ASSIGN_REPLY**

5.27.2 sfs_assign_t Structure

```
typedef struct sfs_assign_s {
  sdd_baseMessage_t base;
  char name[SC_NAME_MAX + 1];
  int readonly;
} sfs_assign_t;
```

5.27.3 Filesystem root object

5.27.3.1 Description

This message is used to add a block device to the filesystem.

The user process sends an **SFS_ASSIGN** request message to the controller process of a filesystem root. The message includes a name of a block device. The controller sends **SFS_ASSIGN_REPLY** reply message back.

5.27.3.2 Structure Members

base	Root filesystem object descriptor.
	Specifies a root filesystem object descriptor.
name	Name of a block device.
	Specifies a name of a block device which shall be retrieved from device manager and added to the filesystem.
readonly	Read-only mode.
	If non-zero, a block device is opened in read-only mode.

5.27.3.3 Errors

base.error	Error code.
EFSALRDYASSIGNED	Device of the same name already assigned.
EFSDRVPROCEXITED	Drive process exited abnormally.
EFSEMPTYNAME	Device name is empty.
EFSINVALNAME	Device name is corrupted.
EFSINVALTYPE	Type of object being added as a drive is invalid.
EFSNODEVMAN	Device manager not found.
EFSNOKERNELD	Could not find kernel daemon.

EFSNOTROOT	Object is not a filesystem root.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSTOOMANYDRIVES	No more space for another drive.
EFSZEROSECTORDEV	Device has got zero sectors.
ENAMETOOLONG	Device name is too long.
ENODEV	Drive initialization failed.
ENOMEM	Out of memory.
EPROCLIM	Cannot create physical drive process.

5.27.4 Physical drive objects

This message is not supported by physical drive objects.

5.28 SFS_FDISK / SFS_FDISK_REPLY

5.28.1 Filesystem root object

This message is not supported by filesystem root object.

5.28.2 Physical drive objects

5.28.2.1 Description

This message is used to partition a physical drive.

The user process sends an **SFS_FDISK** request message to the controller process of a physical drive. The message contains a filesystem specific informations about partitioning. The controller process replies with an **SFS_FDISK_REPLY** reply message.

5.28.2.2 Message IDs

Request message	**SFS_FDISK**
Reply message	**SFS_FDISK_REPLY**

5.28.2.3 sfs_fdisk_t Structure

```
typedef struct sfs_fdisk_s {
  sdd_baseMessage_t base;
  int custom;
} sfs_fdisk_t;
```

5.28.2.4 Structure Members

base	Physical drive object descriptor.
	Specifies a physical drive to partition.
custom	Filesystem specific parameters for partitioning.

5.28.2.5 Partitioning parameters

For FAT filesystem the partitioning parameters are defined as a structure:

```
typedef struct chanfs_fdisk_s {
  uint32_t sizes[4];
} chanfs_fdisk_t;
```

where:

sizes	Partition map table.
	Four items array which specifies how to divide the physical drive. The first item specifies the size of first primary partition and fourth item specifies the fourth primary partition. If the value is less than or equal to 100, it specifies percentage of the partition in the entire disk space. If it is larger than 100, it specifies the par-

tition size in unit of sector.

5.28.2.6 Errors

base.error	Error code.
EBUSY	Cannot partition a drive with mounted partitions.
EFSDRIVEREMOVED	Drive has been removed.
EFSINVALPARAM	Invalid parameter for partitioning a drive.
EFSMSGTOOSMALL	Message size is too small.
EFSNOTDRIVE	Object is not a drive.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSRODRIVE	The physical drive is write protected.
EIO	Error occurred in the low level disk I/O layer.
ENOMEM	Out of memory.
ENOTSUP	Filesystem configuration does not support partitioning.

5.29 SFS_FFLUSH / SFS_FFLUSH_REPLY

5.29.1 Filesystem root object

This message is not supported by filesystem root object.

5.29.2 Physical drive objects

5.29.2.1 Description

This message is used to flush cached information of writing to a disk/partition/file.

The user process sends an SFS_FFLUSH request message to the controller process of a disk/partition/file object. The controller process replies with an SFS_MOUNT_REPLY reply message.

5.29.2.2 Message IDs

Request message	**SFS_FFLUSH**
Reply message	**SFS_FFLUSH_REPLY**

5.29.2.3 sfs_fflush_t Structure

```
typedef struct sfs_fflush_s {
  sdd_baseMessage_t base;
} sfs_fflush_t;
```

5.29.2.4 Structure Members

base	Disk/partition/file object descriptor.
	Specifies a disk/partition/file object which cached informations shall be flushed to the physical medium.

5.29.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSNODEVMAN	Device manager not found.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	Disk or partition or file is not opened.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Error occurred in the low level disk I/O layer.

5.30 SFS_FFIRST / SFS_FFIRST_REPLY

5.30.1 Filesystem root object

This message is not supported by filesystem root object.

5.30.2 Physical drive objects

5.30.2.1 Description

This message is used to find first object in a directory which matches the name pattern.

The user process sends an **SFS_FFIRST** request message to the controller process of a directory object. The controller process replies with an **SFS_FFIRST_REPLY** reply message.

If filesystem is configured for Long File Names support (refer to 3.13 for details), both names of the item, short and long (if exist), are tested.

. never matches any name without extension.

Any pattern terminated with a period never matches any name.

5.30.2.2 Message IDs

Request message	**SFS_FFIRST**
Reply message	**SFS_FFIRST_REPLY**

5.30.2.3 sfs_ffirst_t Structure

```
typedef struct sfs_ffirst_s {
  sdd_obj_t object;
  char pattern[1];
} sfs_ffirst_t;
```

5.30.2.4 Structure Members

object	Directory object descriptor.
	Specifies a directory to search in. In reply message describes found item.
pattern	Name pattern to search for.
	Specifies a pattern of an object name to search for.

5.30.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).

EFSINVALNAME	Name is invalid.
EFSINVALPATTERN	Pattern is corrupted.
EFSISFILE	Object is a file, not directory.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSLOCKED	The operation is rejected according to the file/directory sharing policy.
EFSMSGTOOSMALL	Message size is too small.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMNYOPNFLS	Too many files are opened simultaneously.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	Name is too long to fit into message. Alternatively: File or path name too long.
ENOENT	Item not found in a directory.
ENOMEM	Out of memory.
ENOTSUP	Finding an object on a drive root level is not supported.

5.31 SFS_FNEXT / SFS_FNEXT_REPLY

5.31.1 Filesystem root object

This message is not supported by filesystem root object.

5.31.2 Physical drive objects

5.31.2.1 Description

This message is used to find next object in a directory which matches the name pattern used by SFS_FFIRST message (See 5.30).

The user process sends an **SFS_FNEXT** request message to the controller process of a directory object. The controller process replies with an **SFS_FNEXT_REPLY** reply message.

5.31.2.2 Message IDs

Request message	**SFS_FNEXT**
Reply message	**SFS_FNEXT_REPLY**

5.31.2.3 sfs_fnext_t Structure

```
typedef struct sfs_fnext_s {
  sdd_obj_t object;
} sfs_fnext_t;
```

5.31.2.4 Structure Members

object	Directory object descriptor.
	Specifies a directory to search in. In reply message describes found item.

5.31.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSFNDFSTNOTCALL	SFS_FFIRST was not called.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSISFILE	Object is a file, not directory.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOTFSOBJECT	Not filesystem object.
EIO	Error occurred in the low level disk I/O layer.

ENAMETOOLONG	Name is too long to fit into message.
ENOENT	Item not found in a directory.
ENOMEM	Out of memory.
ENOTSUP	Finding an object on a drive root level is not supported.

5.32 SFS_FGETS / SFS_FGETS_REPLY

5.32.1 Filesystem root object

This message is not supported by filesystem root object.

5.32.2 Physical drive objects

5.32.2.1 Description

This message is used to get a string from a file.

The user process sends an **SFS_FGETS** request message to the reader process of a file object. The reader process replies with an **SFS_FGETS_REPLY** reply message.

The read operation continues until a `'\n'` is stored, reached end of the file or the buffer is filled with **size - 1** characters. The read string is terminated with a `'\0'`.

5.32.2.2 Message IDs

Request message	**SFS_FGETS**
Reply message	**SFS_FGETS_REPLY**

5.32.2.3 sfs_fgets_t Structure

```
typedef struct sfs_fgets_s {
  sdd_baseMessage_t base;
  size_t size;
  char *outlineBuf;
  char inlineBuf[1];
} sfs_fgets_t;
```

5.32.2.4 Structure Members

base	File object descriptor.
	Specifies an opened file object to get a string from.
size	Number of characters to get.
	Specifies a number of characters to get including terminating null character. In return message specifies number of characters stored in the buffer, including terminating null character.
outlineBuf	Pointer to a referenced buffer to store a string to.
<readptr>	Used by the **reply message** and can contain a pointer to the buffer to put the string to. Not recommended as pointers should not be used in messages. Rather use **inlineBuf**.
0	The member **inlineBuf** is used.
inlineBuf	In-message buffer to store a string to.
	Buffer used by the **reply message** if **outlineBuf** is not used. The size is variable and the whole string including terminating null character will be put into this buf-

fer. The size of allocated SFS_FGETS message must be big enough to fit the requested string size.

5.32.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSISDRIVE	Object is a drive, not file.
EFSISRAWOBJECT	Object is a disk or partition for raw access, not file.
EFSMSGTOOSMALL	Message size is too small.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSZEROSIZE	Size of string to get cannot be zero.
EIO	Error occured during read operation.
ENOTSUP	Filesystem configuration does not support this operation.

5.33 SFS_FORMAT / SFS_FORMAT_REPLY

5.33.1 Filesystem root object

This message is not supported by filesystem root object.

5.33.2 Physical drive objects

5.33.2.1 Description

This message is used to create a filesystem on a physical drive.

The user process sends an **SFS_FORMAT** request message to the controller process of a physical drive. The message contains a filesystem specific informations about formatting. The controller process replies with an **SFS_FORMAT_REPLY** reply message.

5.33.2.2 Message IDs

Request message	**SFS_FORMAT**
Reply message	**SFS_FORMAT_REPLY**

5.33.2.3 sfs_format_t Structure

```
typedef struct sfs_format_s {
  sdd_baseMessage_t base;
  int custom;
} sfs_format_t;
```

5.33.2.4 Structure Members

base	Physical drive object descriptor. Specifies a physical drive which contains a partition to format.
custom	Filesystem specific parameters for formatting.

5.33.2.5 Formatting parameters

For FAT filesystem the formatting parameters are defined as a structure:

```
typedef struct chanfs_format_s {
  int partitionToFormat;
  int clusterSize;
} chanfs_format_t;
```

For SAFE FAT filesystem the formatting parameters are defined as a structure:

```
typedef struct chanfs_format_s {
  int partitionToFormat;
  int clusterSize;
  int nonSafeKey;
} chanfs_format_t;
```

where:

partitionToFormat	Specifies a partition number to format.
0	Format entire physical drive.
1-4	Format specific partition on a partitioned physical drive.

clusterSize	Cluster size.
	Specifies a size of the allocation unit (cluster) in unit of byte. The value must be sector size * n (n is 1 to 128 and power of 2). When zero is given, the cluster size is determined depending on the volume size.

nonSafeKey	Non safe format key.
	If this member is set to CHANFS_FORMAT_NON_SAFE_KEY, non-SAFE FAT filesystem will created.

5.33.2.6 Errors

base.error	Error code.
EBUSY	Cannot format mounted partition or partition to format is being accessed raw.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSMSGTOOSMALL	Message size is too small.
EFSNOBOOTSECTOR	Boot sector not found on drive.
EFSNOTDRIVE	Object is not a drive.
EFSNOTVALIDPART	Partition is not valid.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
EFSPARTTOOSMALL	Partition is too small to create filesystem.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSRODRIVE	The physical drive is write protected.
EIO	Error occurred in the low level disk I/O layer.
ENOMEM	Out of memory.
ENOTSUP	Multi partitioning not supported in a current configuration or filesystem configuration does not support formatting.

5.34 SFS_FPUTC / SFS_FPUTC_REPLY

5.34.1 Filesystem root object

This message is not supported by filesystem root object.

5.34.2 Physical drive objects

5.34.2.1 Description

This message is used to put a character to a file.

The user process sends an **SFS_FPUTC** request message to the writer process of a file object. The writer process replies with an **SFS_FPUTC_REPLY** reply message. The reply contains a number of character put.

New line character handling changes depending on a value of **`USE_STRFUNC`** macro:

- When it is set to 1, `'\n'` character is printed without any change.
- When it is set to 2, `'\n'` character is converted to `'\r'+'\n'`.

5.34.2.2 Message IDs

Request message	**SFS_FPUTC**
Reply message	**SFS_FPUTC_REPLY**

5.34.2.3 sfs_fputc_t Structure

```
typedef struct sfs_fputc_s {
  sdd_baseMessage_t base;
  char c;
  size_t size;
} sfs_fputc_t;
```

5.34.2.4 Structure Members

base	File object descriptor.
	Specifies an opened file object to put a character to.
c	Character to put.
size	Number of characters put.
	In **reply message** this field contains a number of characters put to the file.

5.34.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSISDRIVE	Object is a drive, not file.

EFSISRAWOBJECT	Object is a disk or partition for raw access, not file.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Disk full or error occured during write operation.
ENOTSUP	Filesystem configuration does not support this operation.

5.35 SFS_FPUTS / SFS_FPUTS_REPLY

5.35.1 Filesystem root object

This message is not supported by filesystem root object.

5.35.2 Physical drive objects

5.35.2.1 Description

This message is used to put a string to a file.

The user process sends an **SFS_FPUTS** request message to the writer process of a file object. The writer process replies with an **SFS_FPUTS_REPLY** reply message. The reply contains a number of character put.

New line character handling changes depending on a value of **`USE_STRFUNC`** macro:

- When it is set to 1, `'\n'` character is printed without any change.
- When it is set to 2, `'\n'` character is converted to `'\r'+'\n'`.

5.35.2.2 Message IDs

Request message	**SFS_FPUTS**
Reply message	**SFS_FPUTS_REPLY**

5.35.2.3 sfs_fputs_t Structure

```
typedef struct sfs_fputs_s {
  sdd_baseMessage_t base;
  size_t size;
  const char *outlineStr;
  char inlineStr[1];
} sfs_fputs_t;
```

5.35.2.4 Structure Members

base	File object descriptor.
	Specifies an opened file object to put a string to.
size	Number of characters put.
	In reply message this field contains a number of characters put to the file.
outlineStr	Pointer to a referenced string to be put.
<readptr>	Used by the request message and can contain a pointer to the string to put to the file. Not recommended as pointers should not be used in messages. Rather use **inlineStr**. Not used by the reply message and can have any value.
0	The member **inlineBuf** is used.
inlineStr	Included string to be put.
	String to be put used by the request message if **outlineBuf** is not used. The size is variable and the whole string including terminating null character is included.

Not used by the **reply message** and can have any value.

5.35.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSINVALSTRING	Included string is corrupted.
EFSISDRIVE	Object is a drive, not file.
EFSISRAWOBJECT	Object is a disk or partition for raw access, not file.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EIO	Disk full or error occured during write operation.
ENOTSUP	Filesystem configuration does not support this operation.

5.36 SFS_FTELL / SFS_FTELL_REPLY

5.36.1 Filesystem root object

This message is not supported by filesystem root object.

5.36.2 Physical drive objects

5.36.2.1 Description

This message is used to get a current file position pointer.

The user process sends an SFS_FTELL request message to the controller process of a file object (or physical drive or a partition accessed raw). The controller process replies with an SFS_FTELL_REPLY reply message.

5.36.2.2 Message IDs

Request message	**SFS_FTELL**
Reply message	**SFS_FTELL_REPLY**

5.36.2.3 sfs_ftell_t Structure

```
typedef struct sfs_ftell_s {
  sdd_baseMessage_t base;
  uint32_t pos;
} sfs_ftell_t;
```

5.36.2.4 Structure Members

base	File object descriptor.
	Specifies a file object for a position pointer to be returned.
pos	File pointer position.
	Current file pointer position returned by the filesystem.

5.36.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.
ERANGE	Position is out of supported range.

5.37 SFS_FTELL64 / SFS_FTELL64_REPLY

5.37.1 Filesystem root object

This message is not supported by filesystem root object.

5.37.2 Physical drive objects

5.37.2.1 Description

This message is used to get a current file position pointer.

The user process sends an SFS_FTELL64 request message to the controller process of a file object (or physical drive or a partition accessed raw). The controller process replies with an SFS_FTELL64_REPLY reply message.

5.37.2.2 Message IDs

Request message	**SFS_FTELL64**
Reply message	**SFS_FTELL64_REPLY**

5.37.2.3 sfs_ftell64_t Structure

```
typedef struct sfs_ftell64_s {
  sdd_baseMessage_t base;
  uint64_t pos;
} sfs_ftell64_t;
```

5.37.2.4 Structure Members

base	File object descriptor.
	Specifies a file object for a position pointer to be returned.
pos	File pointer position.
	Current file pointer position returned by the filesystem.

5.37.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDRIVEREMOVED	Drive has been removed.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTOPENED	File is not opened.

5.38 SFS_GETSHORTNAME / SFS_GETSHORTNAME_REPLY

5.38.1 Filesystem root object

This message is not supported by filesystem root object.

5.38.2 Physical drive objects

5.38.2.1 Description

This message is used to get a short name of a directory or a file.

The user process sends an **SFS_GETSHORTNAME** request message to the controller process of a directory or a file object. The controller process replies with an **SFS_GETSHORTNAME_REPLY** reply message. The reply message contains a string containing a name in short format.

5.38.2.2 Message IDs

Request message	**SFS_GETSHORTNAME**
Reply message	**SFS_GETSHORTNAME_REPLY**

5.38.2.3 sfs_shortname_t Structure

```
typedef struct sfs_shortname_s {
  sdd_baseMessage_t base;
  char buf[1];
} sfs_shortname_t;
```

5.38.2.4 Structure Members

base	Object descriptor.
	Specifies a directory or a file to get a short name of.
buf	Short name.
	In a **reply message** this field contains a short name of an object. For DOS 8.3 format this field must fit **CHANFS_SHORTNAME_LENGTH** characters.

5.38.2.5 Errors

base.error	Error code.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	Name is invalid.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSMSGTOOSMALL	Message size is too small.

EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSRODRIVE	The physical drive is write protected.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.
ENOTSUP	Cannot get shortname of a drive/partition.

5.39 SFS_MOUNT / SFS_MOUNT_REPLY

5.39.1 Filesystem root object

This message is not supported by filesystem root object.

5.39.2 Physical drive objects

5.39.2.1 Description

This message is used to mount a physical drive or a partition to make it available for the user.

The user process sends an **SFS_MOUNT** request message to the controller process of a physical drive. The message contains a filesystem specific informations about mounting. The controller process replies with an **SFS_MOUNT_REPLY** reply message.

5.39.2.2 Message IDs

Request message	**SFS_MOUNT**
Reply message	**SFS_MOUNT_REPLY**

5.39.2.3 sfs_mount_t Structure

```
typedef struct sfs_mount_s {
  sdd_baseMessage_t base;
  int custom;
} sfs_mount_t;
```

5.39.2.4 Structure Members

base	Physical drive object descriptor.
	Specifies a physical drive to mount or a physical drive which contains a partition to mount.
custom	Filesystem specific parameters for mounting.

5.39.2.5 Mounting parameters

For FAT filesystem the mounting parameters are defined as a structure:

```
typedef struct chanfs_mount_s {
  int partition;
  uint32_t fatCacheSize;
} chanfs_mount_t;
```

For SAFE FAT filesystem the mounting parameters are defined as a structure:

```
typedef struct chanfs_mount_s {
  int partition;
  uint32_t fatCacheSize;
  int nonSafeKey;
} chanfs_mount_t;
```

where:

partition	Partition number to mount.
0	Mount an entire drive. This value can be used in case a physical drive contains a single filesystem.
1-4	Mount a specified partition. This value can be used in case a physical drive is partitioned.
-1	Try to mount all filesystems available on a physical drive.

fatCacheSize	FAT cache size used for this filesystem expressed in number of sectors.
N	FAT cache size is N sectors, or specify CHANFS_MOUNT_USE_DEFAULT_FAT_CACHE_SIZE to use default FAT cache size from configuration file (3.13.6.5.1)

nonSafeKey	Non safe mount key.
	If this member is set to CHANFS_MOUNT_NON_SAFE_KEY, mount will always be in a non-SAFE mode, even if there are valid SAFE FAT structures.

5.39.2.6 Errors

base.error	Error code.
EBUSY	Partition or disk is already mounted or being accessed raw.
EFSCLSTTOOSMALL	Cluster too small to accomodate all entries required for maximum size LFN name.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSMSGTOOSMALL	Message size is too small.
EFSNOBOOTSECTOR	Boot sector not found on drive.
EFSNOFATFS	File system not found on a drive/partition.
EFSNOFILE	Could not find file.
EFSNOTCNSTSAFEFS	Safe filesystem cannot be mounted read-only, because it is inconsistent and must be mounted read-write.
EFSNOTDRIVE	Object is not a drive.
EFSNOTPARTDISK	Drive is not partitioned.
EFSNOTVALIDPART	Partition is not valid.
EFSNOTVLDSAFEFS	There is no valid SAFE FAT volume.
EFSPARTDISK	Drive is partitioned.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
EIO	Error occurred in the low level disk I/O layer.
ENOMEM	Out of memory.

ENOTSUP	Multi partitioning not supported in the current configuration.

5.40 SFS_MOVE / SFS_MOVE_REPLY

5.40.1 Filesystem root object

This message is not supported by filesystem root object.

5.40.2 Physical drive objects

5.40.2.1 Description

This message is used to rename a directory/file and/or move it to another location within the same partition.

The user process sends an **SFS_MOVE** request message to the controller process of an object to be renamed and/or moved. The request message includes the new name and the new directory to move the object to. The controller sends **SDD_MOVE_REPLY** reply message back.

After successful move/rename operation the object is no longer valid and must be immediately released with **SDD_OBJ_RELEASE**.

5.40.2.2 Message IDs

Request message	**SFS_MOVE**
Reply message	**SFS_MOVE_REPLY**

5.40.2.3 sfs_move_t Structure

```
typedef struct sfs_move_s {
  sdd_baseMessage_t base;
  void *newman;
  char newname[SC_NAME_MAX + 1];
} sfs_move_t;
```

5.40.2.4 Structure Members

base	Object descriptor.
	In request message this field contains a descriptor of an object (directory or file) to be moved and/or renamed.

newman	New location for an object.
NULL	Do not move the object.
<handle>	Handle of a new directory to move the object to.

newname	New object name.
	Contains a new object name if the name shall be changed. If it is not required to change the name, it should be the same as the current name.

5.40.2.5 Errors

base.error	Error code.

EACCES	Access denied.
EEXIST	Directory or file already exists.
EFSCORRUPTEDOBJ	Object is corrupted.
EFSDIRFULL	Directory table full - cannot accomodate more entries, disk space still available.
EFSDISKFULL	Disk is full, no more space available.
EFSDRIVEREMOVED	Drive has been removed.
EFSEMPTYNAME	New name string is empty.
EFSINTERNAL	File system structure error or work area broken (possible stack overflow).
EFSINVALNAME	New name string is corrupted. Alternatively: Name is invalid.
EFSISDRIVE	Cannot change drive name.
EFSISPARTITION	Cannot change partition name.
EFSISRAWOBJECT	Object is disk or partition for raw access.
EFSNOFILE	Could not find file.
EFSNOPATH	Could not find path.
EFSNOTFSOBJECT	Not filesystem object.
EFSNOTSAMEPART	Source and destination are not on the same partition.
EFSRDONLYCONFIG	Current filesystem configuration is read-only.
EFSRODRIVE	The physical drive is write protected.
EFSTOOMANYCOLL	Too many name collisions.
EIO	Error occurred in the low level disk I/O layer.
ENAMETOOLONG	File or path name too long.
ENOMEM	Out of memory.

5.41 SFS_UMOUNT / SFS_UMOUNT_REPLY

5.41.1 Filesystem root object

This message is not supported by filesystem root object.

5.41.2 Physical drive objects

5.41.2.1 Description

This message is used to unmount a physical drive or a partition.

The user process sends an **SFS_UMOUNT** request message to the controller process of a physical drive. The message contains a filesystem specific informations about unmounting. The controller process replies with an **SFS_MOUNT_REPLY** reply message.

5.41.2.2 Message IDs

Request message	**SFS_UMOUNT**
Reply message	**SFS_UMOUNT_REPLY**

5.41.2.3 sfs_umount_t Structure

```
typedef struct sfs_umount_s {
  sdd_baseMessage_t base;
  int custom;
} sfs_umount_t;
```

5.41.2.4 Structure Members

base	Physical drive object descriptor.
	Specifies a physical drive to unmount or a physical drive which contains a partition to unmount.
custom	Filesystem specific parameters for unmounting.

5.41.2.5 Unmounting parameters

For FAT filesystem the unmounting parameters are defined as a structure:

```
typedef struct chanfs_umount_s {
  int partition;
} chanfs_umount_t;
```

where:

partition	Partition number to unmount.
1-4	Unmount specified partition. In case a physical drive with single filesystem is mounted (no partitions), value of 1 must be used.
-1	Unmount all mounted filesystems mounted on the physical drive.

5.41.2.6 Errors

base.error	Error code.
EBUSY	Partition/disk is busy with opened files/directories.
EFSDRIVEREMOVED	Drive has been removed.
EFSMSGTOOSMALL	Message size is too small.
EFSNOTDRIVE	Object is not a drive.
EFSPARTOUTOFRNG	Partition number is out of supported range in the current configuration.
ENOENT	Partition/disk is not mounted.

5.42 SFS_UNASSIGN / SFS_UNASSIGN_REPLY

5.42.1 Message IDs

Request message	**SFS_UNASSIGN**
Reply message	**SFS_UNASSIGN_REPLY**

5.42.2 sfs_unassign_t Structure

```
typedef struct sfs_unassign_s {
  sdd_baseMessage_t base;
  char name[SC_NAME_MAX + 1];
  int forcedUnassign;
  int forcedFreeObjects;
} sfs_unassign_t;
```

5.42.3 Filesystem root object

5.42.3.1 Description

This message is used to remove a block device from the filesystem.

The user process sends an **SFS_UNASSIGN** request message to the controller process of a filesystem root. The message includes a name of a block device to be removed. The controller sends **SFS_UNASSIGN_REPLY** reply message back.

5.42.3.2 Structure Members

base	Root filesystem object descriptor.
	Specifies a root filesystem object descriptor.
name	Name of a block device.
	Specifies a name of a block device which shall be removed from the filesystem.
forcedUnassign	Forced unassign.
	If non-zero, a forced block device removal is requested. Refer to 3.6 for more information about removing block device from the filesystem.
forcedFreeObjects	Forced objects freeing.
	If non-zero, a forced objects freeing is requested. When this option is used, application does not have to use **SDD_OBJ_RELEASE** (5.20), to free objects (files, directories, partitions, drive) associated with the device being unassigned. Refer to 3.6 for more information about removing block device from the filesystem.

5.42.3.3 Errors

base.error	Error code.
EACCES	Only a requestor of forced unassign can free objects.
EBUSY	Drive is busy with another removal request or drive is busy with mounted partition or forced removal is already in progress.

EFSDRVPROCEXITED	Drive process exited abnormally.
EFSEMPTYNAME	Drive name is empty.
EFSINVALNAME	Drive name is corrupted.
EFSNOTASSIGNED	Drive was not assigned to the filesystem.
EFSNOTFCDUNASSIGN	Filesystem objects can be freed only with forced unassign.
EFSNOTROOT	Object is not a filesystem root.
EFSOBJSNOTFREED	Cannot unassign due to not freed objects.
ENOMEM	Out of memory.

5.42.4 Physical drive objects

This message is not supported by physical drive objects.

6 Application Programmer Interface

6.1 Introduction

In this chapter all SCIOPTA File System functions are described.

The functions are listed in alphabetical order.

6.1.1 Header file

The function prototypes are defined in the following header file:

<installation_folder>\sciopta\<version>\include\sfs\sfs.h

6.1.2 Kernel libraries

The following API functions are included in the kernel libraries. The kernel libraries are build for the suppor-ted compilers and can be found here:

<installation_folder>\sciopta\<version>\lib\<arch>\

Please consult the kernel libraries sections of the "Building SCIOPTA Systems" chapter of the SCIOPTA - Real-Time Kernel User's Manual for more information about kernel libraries.

6.1.3 Source code

Instead of using the sfs libraries you could include the source code in your project. This would be useful if other compiler switches are needed as the ones used for building the libraries.

The source codes of all API functions are supplied in the SCIOPTA kernel delivery and can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\

6.2 sfs_assign

6.2.1 Description

The **sfs_assign** function is used to add a block device to the filesystem.

The function sends an **SFS_ASSIGN** message to the controller process of a filesystem root object and waits for an **SFS_ASSIGN_REPLY** message. The calling process will be blocked until this message is received.

6.2.2 Syntax

```
int sfs_assign (
   const sdd_obj_t * root,
   const char name[],
   int readonly
);
```

6.2.3 Parameter

root	Root filesystem object descriptor.
	Specifies a root filesystem object to which a new drive shall be added.
name	Name of a block device.
	Specifies a name of a block device which shall be retrieved from device manager and added to the filesystem.
readonly	Read-only mode.
	If non-zero, a block device is opened in read-only mode.

6.2.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.2.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_ASSIGN** (5.27.3.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_assign** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.2.6 Source Code

The source code of the **sfs_assign** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_assign.c

6.2.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * drive;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_assign(chanFs, "ramdisk0", 0) < 0) {
  PRINTF ("Error assigning device to filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
if (drive == NULL) {
  PRINTF ("Could not get physical drive object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.3 sfs_clearerr

6.3.1 Description

The **sfs_clearerr** function is used to clear error indicator of a file, end-of-file indicator of a file and process errno variable.

Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.3.2 Syntax

```
int sfs_clearerr (
   sdd_obj_t *file
);
```

6.3.3 Parameter

file	Descriptor of a file.
	Specifies a file for which to clear error indicator and end-of-file inidicator.

6.3.4 Return Value

No value is returned by this function.

If file descriptor pointer passed to this function is NULL, process errno variable will be set to EFSNULLPOINTER. Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.3.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_clearerr** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.3.6 Source Code

The source code of the **sfs_clearerr** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_clearerr.c

6.3.7 Example

```
  .
  .
  .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[1];
size_t ret;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

sfs_clearerr(file);
while (!sfs_feof(file)) {
  ret = sfs_fread(buffer, 1, sizeof(buffer), file);

  if (ret != sizeof(buffer)) {
    if (sfs_feof(file) != 0) {
      PRINTF ("End of file \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
    PRINTF ("Error reading file (%d)\n", sfs_ferror(file));
    sc_procKill (SC_CURRENT_PID, 0);
  }
}

  .
  .
  .
```

6.4 sfs_chdir

6.4.1 Description

The **sfs_chdir** function is used to change a path (current working directory) of a directory object.

The function sends multiple **SDD_MAN_GET** messages to the controller process of a filesystem object and waits for **SDD_MAN_GET_REPLY** messages. The calling process will be blocked until these messages are received.

New directory can be any object up to the **device manager**.

If **path** starts with '/' then **path** will be treated as absolute, but starting from **device manager**. The **path** may contain **'..'** symbols, which refer to an upper directory. Following is an example of a **path** pointing to directory **bar** on a second partition of drive **ramdisk0**.

```
Newdir = sfs_chdir(dir, "/sdd_chanfs/ramdisk0/p2/foo/bar");
```

If **path** does not start with **'/'** then **path** is treated as a relative to object **dir**. The **path** may contain **'..'** symbols, which refer to an upper directory. Following is an example of a **path** pointing to the parent directory of the object **dir**.

```
newdir = sfs_chdir(dir, "..");
```

or

```
newdir = sfs_chdir(dir, "foo/../..");
```

6.4.2 Syntax

```
int sfs_chdir (
   sdd_obj_t * dir,
   const char * path
);
```

6.4.3 Parameter

dir	Descriptor of a directory object.
	Speciefies a directory object which path shall be changed.
path	New path.

6.4.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.4.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_get** (6.30.5) and **SDD_OBJ_DUP** (5.19.3.3, 5.19.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_chdir** call.

EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted anymore.
EFSNULLPOINTER	Directory is NULL or new path is NULL.
EINVAL	Directory type is not 'directory' or 'manager' type.
ENOMEM	Out of memory.

6.4.6 Source Code

The source code of the **sfs_chdir** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_chdir.c

6.4.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* testfile.txt file is located in /ramdisk0/p2/testdir */

  if (sfs_chdir(chanFs, "/ramdisk0/p2/testdir") < 0) {
    PRINTF ("Could not change directory \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.5 sfs_copy

6.5.1 Description

The **sfs_copy** function is used to rename a directory/file and/or move it to another location within the same partition or between different partitions and disks.

The function sends multiple messages to the controller process of a filesystem object and waits for reply messages. The calling process will be blocked until these messages are received.

The **sfs_copy** function uses only paths, which are relative to **root** object. For the format of relative paths, refer to 3.4.2.

When using **sfs_copy**, following rules apply:

- If **oldpath** specifies a directory and **newpath** specifies a directory which does not exist: **sfs_copy** creates the new directory and copies **only the content of the old directory** to the newly created one.
- If **oldpath** specifies a directory and **newpath** specifies a directory which already exists: **sfs_copy** copies an **old directory including its content** to the new directory.

The **sfs_copy** is a function which uses recursion. The maximum depth of recursion is specified by macro define **SFS_DIR_CPY_MAX_DEPTH** in source code file of this function (refer to 6.5.6 for location of the source code). Default maximum depth level is set to 10. If different level is required, change the macro definition and rebuild Sciopta File System library.

6.5.2 Syntax

```
int sfs_copy (
   sdd_obj_t * root,
   const char * oldpath,
   const char * newpath
);
```

6.5.3 Parameter

root	Filesystem object descriptor.
	Specifies a filesystem directory object.
oldpath	Old path.
	Specifies a path to the object which shall be copied.
newpath	New path.
	Specifies a new path for the object.

6.5.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.5.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_getcwd** (6.31.5), **sfs_create** (6.6.5), **SDD_DEV_OPEN** (5.5.2.5), **SDD_DEV_READ** (5.6.2.5),

SDD_DEV_WRITE (5.7.2.5) and **SDD_MAN_GET** (5.13.3.3, 5.13.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_copy** call.
EEXIST	Paths are the same or file already exists.
EFSEMPTYNAME	Path cannot be empty.
EFSINVALNAME	Path is invalid.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted any-more.
EFSNULLPOINTER	Parameter 'root' is NULL or path is NULL.
EFSTOODEEP	Maximum objects depth reached.
EINVAL	Not-last segment of path is a file.
ENAMETOOLONG	New path is too long or old path is too long or recursive path is too long.
ENOENT	Old path not found or new path not found.

6.5.6 Source Code

The source code of the **sfs_copy** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_move.c

6.5.7 Example

```
sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * directory;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_copy(chanFs, "/ramdisk0/p2/testdir", "/ramdisk0/p4") < 0) {
  PRINTF ("Could not copy directory and its content \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

directory = sfs_get(chanFs, "/ramdisk0/p4/testdir", SFS_ATTR_DIR);
if (directory == NULL) {
  PRINTF ("Could not get directory object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}
```

6.6 sfs_create

6.6.1 Description

The **sfs_create** function is used to create a new directory or a file object.

The function sends an **SDD_MAN_ADD** message to the controller process of a partition or a directory object and waits for an **SDD_MAN_ADD_REPLY** message. The calling process will be blocked until this message is received.

If **dir** reference object is **not** specified, an absolute path must be used. For the format of absolute paths refer to 3.4.1.

If **dir** reference object is specified, a relative path must be used. For the format of relative paths refer to 3.4.2.

6.6.2 Syntax

```
int sfs_create (
   sdd_obj_t * dir,
   const char * name,
   sc_msgid_t type,
   mode_t mode
);
```

6.6.3 Parameter

dir	Directory object
	Specifies a directory for a new object.
name	Name of new object.
	Specifies a name of a new object to create.
type	Type of object.
	Specifies a type of object to create. **SFS_ATTR_FILE** and **SFS_ATTR_DIR** are the only two valid types.
mode	Not implemented for every FS.

6.6.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.6.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_MAN_ADD** (5.12.4.3) and **SDD_MAN_GET** (5.13.3.3, 5.13.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_create** call.
EBADF	A relative path must have a directory object.
EFSEMPTYNAME	Name cannot be empty.

EFSINVALNAME	Name is invalid or no name specified to create.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted anymore.
EFSNOROOT	Could not get root directory.
EFSNULLPOINTER	Name cannot be NULL.
ENAMETOOLONG	Name is too long.
ENOMEM	Out of memory.

6.6.6 Source Code

The source code of the **sfs_create** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_create.c

6.6.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * partition;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_create(partition, "testdir", SFS_ATTR_DIR, 0) < 0) {
  PRINTF ("Could not create a directory \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_create(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE, 0) < 0) {
  PRINTF ("Could not create a file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.7 sfs_delete

6.7.1 Description

The **sfs_delete** function is used to delete a directory or a file object.

The function sends an **SDD_MAN_RM** message to the controller process of a directory object and waits for an **SDD_MAN_RM_REPLY** message. The calling process will be blocked until this message is received.

This function deletes a directory only if it is empty. For a recursive version of this function refer to **sfs_deleteRecursive** (6.8).

If **root** reference object is **not** specified, an absolute path must be used. For the format of absolute paths refer to 3.4.1.

If **root** reference object is specified, a relative path must be used. For the format of relative paths refer to 3.4.2.

6.7.2 Syntax

```
int sfs_delete (
   sdd_obj_t * root,
   const char * path
);
```

6.7.3 Parameter

root	Filesystem object descriptor.
	Specifies a filesystem directory object.
path	Path of on object to delete.

6.7.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.7.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_get** (6.30.5), **SDD_MAN_GET** (5.13.3.3, 5.13.4.3) and **SDD_MAN_RM** (5.18.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_delete** call.
EBADF	Relative path needs a root directory.
EFSEMPTYNAME	Parameter 'path' cannot be empty.
EFSINVALNAME	Parameter 'path' is invalid or there is nothing to be removed (empty name).
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted anymore.
EFSNOROOT	Parameter 'root' is NULL and manager process not found.

EFSNULLPOINTER	Parameter 'path' is NULL.
ENAMETOOLONG	Parameter 'path' is too long.
ENOMEM	Out of memory.

6.7.6 Source Code

The source code of the **sfs_delete** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_delete.c

6.7.7 Example

```
  .
  .
  .

sdd_obj_t * man;
sdd_obj_t * chanFs;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_delete(chanFs, "/ramdisk0/p2/testdir/testfile.txt") < 0) {
  PRINTF ("Could not delete file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_chdir(chanFs, "/ramdisk0/p2") < 0) {
  PRINTF ("Could not change current directory \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_delete(chanFs, "testdir") < 0) {
  PRINTF ("Could not delete directory \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

  .
  .
  .
```

6.8 sfs_deleteRecursive

6.8.1 Description

The **sfs_deleteRecursive** function is used to delete a file or a directory with its content.

The function sends multiple **SDD_MAN_RM** messages to the controller process of a filesystem object and waits for **SDD_MAN_RM_REPLY** messages. The calling process will be blocked until these messages are received.

For a non-recursive version of this function, refer to **sfs_delete** (6.7).

If **root** reference object is **not** specified, an absolute path must be used. For the format of absolute paths refer to 3.4.1.

If **root** reference object is specified, a relative path must be used. For the format of relative paths refer to 3.4.2.

sfs_deleteRecursive is a function which uses recursion. The maximum depth of recursion is specified by macro define **SFS_DIR_DELETERECURSIVE_MAX_DEPTH** in source code file of this function (refer to 6.8.6 for location of the source code). Default maximum depth level is set to 10. If different level is required, change the macro definition and rebuild Sciopta File System library.

6.8.2 Syntax

```
int sfs_deleteRecursive (
   sdd_obj_t *root,
   const char * path
);
```

6.8.3 Parameter

root	Filesystem object descriptor.
	Specifies a filesystem directory object.
path	Path of on object to delete.

6.8.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.8.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_delete** (6.7.5), **SDD_MAN_GET** (5.13.3.3, 5.13.4.3) and **SDD_MAN_RM** (5.18.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_deleteRecursive** call.
EBADF	Relative path needs a root directory.
EFSEMPTYNAME	Parameter 'path' cannot be empty.
EFSINVALNAME	Parameter 'path' is invalid or there is nothing to be removed (empty name).

EFSNOROOT	Parameter 'root' is NULL and manager process not found.
EFSNULLPOINTER	Parameter 'path' is NULL.
EFSTOODEEP	Maximum objects depth reached.
ENAMETOOLONG	Parameter 'path' is too long.
ENOENT	Path does not exist.

6.8.6 Source Code

The source code of the **sfs_deleteRecursive** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_deleterecursive.c

6.8.7 Example

```
sdd_obj_t * man;
sdd_obj_t * chanFs;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_deleteRecursive(chanFs, "/ramdisk0/p2/testdir") < 0) {
  PRINTF ("Could not delete directory with its content \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

  .
  .
  .
```

6.9 sfs_fclose

6.9.1 Description

The **sfs_fclose** function is used to close an opened disk/partition/file.

The function sends an **SDD_DEV_CLOSE** message to the controller process of an opened disk/partition/file object and waits for an **SDD_DEV_CLOSE_REPLY** message. The calling process will be blocked until this message is received.

6.9.2 Syntax

```
int sfs_fclose (
   sdd_obj_t * file
);
```

6.9.3 Parameter

file	Descriptor of a disk/partition/file to close.
	Specifies an opened disk/partition/file to be closed.

6.9.4 Return Value

If the function succeeds the return value is zero.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.9.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_DEV_CLOSE** (5.3.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fclose** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.9.6 Source Code

The source code of the **sfs_fclose** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fclose.c

6.9.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_fclose(file) < 0) {
  PRINTF ("Could not close file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_free(&file) < 0) {
  PRINTF ("Could not free file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.10 sfs_fdisk

6.10.1 Description

The **sfs_fdisk** function is used to add a block device to the filesystem.

The function sends an **SFS_FDISK** message to the controller process of a physical drive object and waits on an **SFS_FDISK_REPLY** message. The calling process will be blocked until this message is received.

6.10.2 Syntax

```
int sfs_fdisk (
   sdd_obj_t * drive,
   void *custom,
   size_t custom_size
);
```

6.10.3 Parameter

drive	Drive descriptor.
	Specifies a drive descriptor for partitioning.
custom	Partitioning parameters (filesystem dependent).
	Specifies how a drive shall be partitioned.
custom_size	Size of partitioning parameters (filesystem dependent).

6.10.4 Partitioning parameters

For FAT filesystem the partitioning parameters are defined as a structure:

```
typedef struct chanfs_fdisk_s {
  uint32_t sizes[4];
} chanfs_fdisk_t;
```

where:

sizes	Partition map table.
	Four items array which specifies how to divide the physical drive. The first item specifies the size of first primary partition and fourth item specifies the fourth primary partition. If the value is less than or equal to 100, it specifies percentage of the partition in the entire disk space. If it is larger than 100, it specifies the partition size in unit of sector.

6.10.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.10.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FDISK** (5.28.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fdisk** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	Out of memory.

6.10.7 Source Code

The source code of the **sfs_fdisk** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fdisk.c

6.10.8 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * dev;
  sdd_obj_t * drive;
  chanfs_fdisk_t fdisk;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  dev = sdd_manGetByName(man, "ramdisk0");
  if (dev == NULL) {
    PRINTF ("Could not find ramdisk device \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_assign(chanFs, &(dev)) < 0) {
    PRINTF ("Error assigning device to filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
  if (drive == NULL) {
    PRINTF ("Could not get physical drive object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  fdisk.sizes[0] = 20; /* 1st partition occupies 20% of drive */
  fdisk.sizes[1] = 30; /* 2nd partition occupies 30% of drive */
  fdisk.sizes[2] = 25; /* 3rd partition occupies 25% of drive */
  fdisk.sizes[3] = 25; /* 4th partition occupies 25% of drive */
  if (sfs_fdisk(drive, &fdisk, sizeof(fdisk)) < 0) {
    PRINTF ("Error partitioning physical drive \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.11 sfs_feof

6.11.1 Description

The **sfs_feof** function is used to test for an end-of-file on a file indicator. For physical drive or a partition accessed in raw mode, end-of-file indicator is not set. Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.11.2 Syntax

```
int sfs_feof (
   const sdd_obj_t *file
);
```

6.11.3 Parameter

file	Descriptor of a file.
	Specifies a file for which the end-of-file indicator shall be returned.

6.11.4 Return Value

Non-zero value is returned in case end-of-file indicator is detected, otherwise 0 is returned.

If file descriptor pointer passed to this function is NULL, EFSNULLPOINTER error will be returned and process errno variable will be set to EFSNULLPOINTER. If descriptor passed to this function is not a file, EBADF error will be returned and process errno variable will be set to EBADF. Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.11.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_feof** call.
EFSNOTFILE	Descriptor pointer is not a file.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	Out of memory.

6.11.6 Source Code

The source code of the **sfs_feof** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_feof.c

6.11.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[1];
size_t ret;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

sfs_clearerr(file);
while (!sfs_feof(file)) {
  ret = sfs_fread(buffer, 1, sizeof(buffer), file);

  if (ret != sizeof(buffer)) {
    if (sfs_feof(file) != 0) {
      PRINTF ("End of file \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
    PRINTF ("Error reading file (%d)\n", sfs_ferror(file));
    sc_procKill (SC_CURRENT_PID, 0);
  }
}

    .
    .
    .
```

6.12 sfs_ferror

6.12.1 Description

The **sfs_ferror** function is used to get an error indicator of a file. Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.12.2 Syntax

```
int sfs_ferror (
   const sdd_obj_t *file
);
```

6.12.3 Parameter

file	Descriptor of a file.
	Specifies a file for which the end-of-file mark shall be returned.

6.12.4 Return Value

Error code (non-zero value) is returned in case error indicator is set on a file. Otherwise 0 is returned.

If file descriptor pointer passed to this function is NULL, EFSNULLPOINTER error will be returned and process errno variable will be set to EFSNULLPOINTER. If error indicator for a file is not set, process errno variable is returned. Refer to 3.8.12 for details about file error indicator, end-of-file indicator and usage of this function.

6.12.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_ferror** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.12.6 Source Code

The source code of the **sfs_ferror** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_ferror.c

6.12.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[1];
size_t ret;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

sfs_clearerr(file);
while (!sfs_feof(file)) {
  ret = sfs_fread(buffer, 1, sizeof(buffer), file);

  if (ret != sizeof(buffer)) {
    if (sfs_feof(file) != 0) {
      PRINTF ("End of file \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
    PRINTF ("Error reading file (%d)\n", sfs_ferror(file));
    sc_procKill (SC_CURRENT_PID, 0);
  }
}

    .
    .
    .
```

6.13 sfs_fflush

6.13.1 Description

The **sfs_fflush** function is used to flush cached information of writing to a disk/partition/file.

The function sends an **SFS_FFLUSH** message to the controller process of a disk/partition/file object and waits for an **SFS_FFLUSH_REPLY** message. The calling process will be blocked until this message is received.

6.13.2 Syntax

```
int sfs_fflush (
   sdd_obj_t * file
);
```

6.13.3 Parameter

file	Disk/partition/file object descriptor.
	Specifies a disk/partition/file to sync to mounted medium.

6.13.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.13.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FFLUSH** (5.29.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fflush** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate message.

6.13.6 Source Code

The source code of the **sfs_fflush** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fflush.c

6.13.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[10] = {"test text"};
size_t characters_written;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

characters_written = sfs_fwrite(buffer, 1, sizeof(buffer), file);

if (characters_written != sizeof(buffer)) {
  PRINTF ("Error writing file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_fflush(file) < 0) {
  PRINTF ("Error flushing file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.14 sfs_fgets

6.14.1 Description

The **sfs_fgets** function is used to get a string from a file.

The function sends an **SFS_FGETS** message to the reader process of a file object and waits for an **SFS_FGETS_REPLY** message. The calling process will be blocked until this message is received.

The read operation continues until a `'\n'` is stored, reached end of the file or the buffer is filled with **len - 1** characters. The read string is terminated with a `'\0'`.

6.14.2 Syntax

```
char * sfs_fgets (
   char * buff,
   size_t len,
   sdd_obj_t * file
);
```

6.14.3 Parameter

buff	Buffer.
	Specifies a buffer to store a read string to.
len	Size of buffer.
	Specifies a size of a buffer to store read characters (including terminating null character).
file	Descriptor of a file.
	Specifies a file to read from.

6.14.4 Return Value

If the function succeeds, **buff** is returned.

If end of file is reached while attempting to read a character, end-of-file indicator is set.

If there was nothing to read, NULL is returned and content of **buff** remains unchanged.

If error occurred, file error indicator is set and NULL is returned. Content of **buff** may have changed.

To check if end-of-file indicator has been set, use function **sfs_feof** (6.11). To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator and end-of-file indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.14.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FGETS** (5.32.2.5).

error code	Return value of **sc_miscErrnoGet**

	After an **sfs_fgets** call.
EFSNULLPOINTER	File descriptor pointer is NULL or buffer pointer is NULL.
EINVAL	Len must be greater than zero.
ENOMEM	No memory to allocate message.

6.14.6 Source Code

The source code of the **sfs_fgets** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fstr.c

6.14.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[10];
char * ret;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

ret = sfs_fgets(buffer, sizeof(buffer), file);

if (ret == NULL) {
  if (sfs_feof(file) == 1) {
    PRINTF ("No string to read \n");
    sc_procKill (SC_CURRENT_PID, 0);
  } else {
    PRINTF ("Error reading string \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }
}

    .
    .
    .
```

6.15 sfs_findFirst

The **sfs_findFirst** function is used to find a first object in a directory which matches the name pattern.

The function sends an **SFS_FFIRST** message to the controller process of a directory object and waits for an **SFS_FFIRST_REPLY** message. The calling process will be blocked until this message is received.

The matching pattern can contain wildcard characters. A **?** matches an any character and an ***** matches any string of zero length or longer.

If filesystem is configured for Long File Names support (refer to 3.13 for details), both names of the item, short and long (if exist), are tested.

. never matches any name without extension.

Any pattern terminated with a period never matches any name.

6.15.1 Syntax

```
sdd_obj_t * sfs_findFirst (
   const sdd_obj_t * dir,
   char * pattern
);
```

6.15.2 Parameter

dir	Directory object descriptor.
	Specifies a directory to search in.
pattern	Name pattern to search for.
	Specifies a pattern of an object name to search for.

6.15.3 Return Value

If the function succeeds, found object is returned.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.15.4 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FFIRST** (5.30.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_findFirst** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.15.5 Source Code

The source code of the **sfs_findFirst** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_find.c

6.15.6 Example

```
    .
    .
    .

  logd_t *logd;
  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * directory;
  sdd_obj_t * item;
  sdd_obj_t * previous;

  logd = logd_new ("/SCP_logd",
                   LOGD_INFO,
                   "test",
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* get directory object */
  directory = sfs_get(chanFs, "/ramdisk0/p2/testdir", SFS_ATTR_DIR);
  if (directory == NULL) {
    PRINTF ("Could not get directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  item = sfs_findFirst(directory, "testfile*.txt");
  while (item) {
    logd_printf(logd, LOGD_INFO, "Item: %s \n", item->name);

    previous = item;
    item = sfs_findNext(directory);
    if (sfs_free(&previous) < 0) {
      PRINTF ("Could not free object \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
  }

  if (sfs_free(&directory) < 0) {
    PRINTF ("Could not free object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.16 sfs_findNext

The **sfs_fnext** function is used to find a next object in a directory which matches the name pattern used in **sfs_findFirst** function call (see 6.15).

The function sends an **SFS_FNEXT** message to the controller process of a directory object and waits for an **SFS_FNEXT_REPLY** message. The calling process will be blocked until this message is received.

6.16.1 Syntax

```
sdd_obj_t * sfs_findNext (
   const sdd_obj_t * dir
);
```

6.16.2 Parameter

dir	Directory object descriptor.
	Specifies a directory to search in.

6.16.3 Return Value

If the function succeeds, found object is returned.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.16.4 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FNEXT** (5.31.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_findNext** call.
EINVAL	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.16.5 Source Code

The source code of the **sfs_findNext** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_find.c

6.16.6 Example

```
    .
    .
    .

  logd_t *logd;
  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * directory;
  sdd_obj_t * item;
  sdd_obj_t * previous;

  logd = logd_new ("/SCP_logd",
                   LOGD_INFO,
                   "test",
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* get directory object */
  directory = sfs_get(chanFs, "/ramdisk0/p2/testdir", SFS_ATTR_DIR);
  if (directory == NULL) {
    PRINTF ("Could not get directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  item = sfs_findFirst(directory, "testfile*.txt");
  while (item) {
    logd_printf(logd, LOGD_INFO, "Item: %s \n", item->name);

    previous = item;
    item = sfs_findNext(directory);
    if (sfs_free(&previous) < 0) {
      PRINTF ("Could not free object \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
  }

  if (sfs_free(&directory) < 0) {
    PRINTF ("Could not free object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.17 sfs_fopen

6.17.1 Description

The **sfs_fopen** function is used to open a disk/partition/file object for read, write or read/write.

The function sends an **SDD_DEV_OPEN** message to the controller process of a disk/partition/file object and waits for an **SDD_DEV_OPEN_REPLY** message. The calling process will be blocked until this message is received.

The **sfs_fopen** uses a path to specify a file to open. If file object is already present, **sfs_open** (6.41) may be used.

If **dir** reference object is **not** specified, an absolute path must be used. For the format of absolute paths refer to 3.4.1.

If **dir** reference object is specified, a relative path must be used. For the format of relative paths refer to 3.4.2.

6.17.2 Syntax

```
sdd_obj_t * sfs_fopen (
   sdd_obj_t * dir,
   const char * path,
   const char * mode
);
```

6.17.3 Parameter

dir	Descriptor of a starting point object.
	Specifies an object to start the search from.
path	Path of an object disk/partition/file to open.
mode	Open mode.
	Specifies flags to be used when opening the disk/partition/file.
“r”	Open file for read operations. File must exist.
“wo”	Open file for write operations. File must exist.
“w”	Create an empty file for write operations. If a file with the same name already exists, its content is discarded and the file is treated as a new empty file.
“a”	Open file for write operations at the end of file. Write operations always write data at the end of file, expanding it. Seek and resize operations are not possible and will return error. If file does not exist, it will be created.
“r+”	Open file for read and write operations. File must exist.
“w+”	Create an empty file for read and write operations. If a file with the same name already exists, its content is discarded and the file is treated as a new empty file.
“a+”	Open file for both read and write operations with all write operations at the end of file. Write operations always write data at the end of file, expanding it. Seek and resize operations are possible, but any subsequent moves file pointer position to the end of file prior and after writing. If file does not exist, it will be created.

6.17.4 Return Value

If the function succeeds the return value is an opened disk/partition/file object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.17.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_create** (6.6.5), **sfs_get** (6.30.5) and **SDD_DEV_OPEN** (5.5.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fopen** call.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted any-more.
EFSNULLPOINTER	Parameter is NULL.
EINVAL	Mode specified is invalid.

6.17.6 Source Code

The source code of the **sfs_fopen** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_open.c

6.17.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
  if (file == NULL) {
    PRINTF ("Could not open file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.18 sfs_format

6.18.1 Description

The **sfs_format** function is used to create a filesystem on a physical drive.

The function sends an **SFS_FORMAT** message to the controller process of a physical drive object and waits for an **SFS_FORMAT_REPLY** message. The calling process will be blocked until this message is received.

6.18.2 Syntax

```
int sfs_format (
   sdd_obj_t * drive,
   void *custom,
   size_t custom_size
);
```

6.18.3 Parameter

drive	Drive descriptor.
	Specifies a drive descriptor for formatting.
custom	Partitioning parameters (filesystem dependent).
	Specifies which partition shall be formatted.
custom_size	Size of formatting parameters (filesystem dependent).

6.18.4 Formatting parameters

For FAT filesystem the formatting parameters are defined as a structure:

```
typedef struct chanfs_format_s {
  int partitionToFormat;
  int clusterSize;
} chanfs_format_t;
```

For SAFE FAT filesystem the formatting parameters are defined as a structure:

```
typedef struct chanfs_format_s {
  int partitionToFormat;
  int clusterSize;
  int nonSafeKey;
} chanfs_format_t;
```

where:

partitionToFormat	Specifies a partition number to format.
0	Format entire physical drive.
1-4	Format specific partition on a partitioned physical drive.
clusterSize	Cluster size.
	Specifies a size of the allocation unit (cluster) in unit of byte. The value must be sector size * n (n is 1 to 128 and power of 2). When zero is given, the cluster size is determined depending on the volume size.

nonSafeKey	Non safe format key.
	If this member is set to CHANFS_FORMAT_NON_SAFE_KEY, non-SAFE FAT filesystem will created.

6.18.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.18.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FORMAT** (5.33.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_format** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.18.7 Source Code

The source code of the **sfs_format** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_format.c

6.18.8 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * drive;
chanfs_format_t format;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
if (drive == NULL) {
  PRINTF ("Could not get physical drive object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* Drive is already partitioned. Format 2nd partition */

format.clusterSize = 8192;
format.partitionToFormat = 2;
if (sfs_format(drive, &format, sizeof(format)) < 0) {
  PRINTF ("Error formatting partition \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
```

.
.

6.19 sfs_fputc

6.19.1 Description

The **sfs_fputc** function is used to put a character to a file.

The function sends an **SFS_FPUTC** message to the writer process of a file object and waits for an **SFS_FPUTC_REPLY** message. The calling process will be blocked until this message is received.

New line character handling changes depending on a value of **`USE_STRFUNC`** macro:

- When it is set to 1, `'\n'` character is printed without any change.
- When it is set to 2, `'\n'` character is converted to `'\r'+'\n'`.

6.19.2 Syntax

```
int sfs_fputc (
   char character,
   sdd_obj_t * file
);
```

6.19.3 Parameter

character	Character.
	Specifies a character to write.
file	Descriptor of a file.
	Specifies a file to write a character to.

6.19.4 Return Value

If the function succeeds **character** is returned (**character** is cast to 8-bit unsigned value and then cast to **int**).

If the function fails, the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.19.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FPUTC** (5.34.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fputc** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate message.

6.19.6 Source Code

The source code of the **sfs_fputc** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fstr.c

6.19.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_fputc('a', file) < 0) {
    PRINTF ("Error putting character to file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.20 sfs_fputs

6.20.1 Description

The **sfs_fputs** function is used to put a string to a file.

The function sends an **SFS_FPUTS** message to the writer process of a file object and waits for an **SFS_FPUTS_REPLY** message. The calling process will be blocked until this message is received.

New line character handling changes depending on a value of **`USE_STRFUNC`** macro:

- When it is set to 1, `'\n'` character is printed without any change.
- When it is set to 2, `'\n'` character is converted to `'\r'+'\n'`.

6.20.2 Syntax

```
int sfs_fputs (
   const char *str,
   sdd_obj_t * file
);
```

6.20.3 Parameter

str	String
	Specifies a string to write.
file	Descriptor of a file.
	Specifies a file to write a string to.

6.20.4 Return Value

If the function succeeds non-negative value is returned.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.20.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FPUTS** (5.35.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fputs** call.
EFSNULLPOINTER	File descriptor pointer is NULL or buffer pointer is NULL.
ENOMEM	No memory to allocate message.

6.20.6 Source Code

The source code of the **sfs_fputs** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fstr.c

6.20.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_fputs("randomtext", file) < 0) {
    PRINTF ("Error putting string to file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.21 sfs_free

6.21.1 Description

The **sfs_free** function is used to release and free a filesystem object representing a physical drive, partition, directory or a file.

The function sends an **SDD_OBJ_RELEASE** message to the controller process of a filesystem object and waits for an **SDD_OBJ_RELEASE_REPLY** message. The calling process will be blocked until this message is received.

Every object returned by filesystem functions must be freed by using **sfs_free**. Refer to 3.10 for details about freeing filesystem objects.

6.21.2 Syntax

```
int sfs_free (
   sdd_obj_t **obj
);
```

6.21.3 Parameter

obj	Descriptor of an object.
	Specifies an object to be freed. **If function succeeds, the object pointer is set to NULL and object can no longer be used.**

6.21.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.21.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_RELEASE** (5.20.3.3, 5.20.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_free** call.
EFSNULLPOINTER	Parameter is NULL or points to a NULL pointer.

6.21.6 Source Code

The source code of the **sfs_free** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_free.c

6.21.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  ...

  << operation on a file object >>

  ...

  if (sfs_free(&file) < 0) {
    PRINTF ("Could not free file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.22 sfs_fread

6.22.1 Description

The **sfs_fread** function is used to read data from an opened disk/partition/file.

The function sends an **SDD_DEV_READ** message to the reader process of a disk/partition/file object and waits for an **SDD_DEV_READ_REPLY** message. The calling process will be blocked until this message is received.

6.22.2 Syntax

```
size_t sfs_fread (
   void * buf,
   size_t size,
   size_t count,
   sdd_obj_t * file
);
```

6.22.3 Parameter

buf	Buffer.
	Specifies a buffer to store the read data to.
size	Size of each element to read.
	Specifies size in bytes of each elements that will be stored in the buffer.
count	Number of elements to read.
	Specifies number of elements to read, each of size of **size** bytes.
file	Descriptor of a disk/partition/file.
	Specifies a disk/partition/file to read from.

6.22.4 Return Value

Function returns number of elements successfully read from a disk/partition/file.

If either **size** or **count** parameter is zero, the function returns zero and state of file does not change.

If return value is different than parameter **count**, either read error occurred or end-of-file has been reached. To check if end-of-file indicator has been set, use function **sfs_feof** (6.11). To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator and end-of-file indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.22.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_DEV_READ** (5.6.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fread** call.

EFSNULLPOINTER	File descriptor pointer is NULL or buffer pointer is NULL.
ERANGE	Number of bytes to read exceeds parameter type range.

6.22.6 Source Code

The source code of the **sfs_fread** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fread.c

6.22.7 Example

```
  .
  .
  .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[10];
size_t characters_read;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

characters_read = sfs_fread(buffer, 1, sizeof(buffer), file);

if ( characters_read != sizeof(buffer) ) {
  if (sfs_feof(file) > 0) {
    PRINTF ("Not enough data to read in file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  } else {
    PRINTF ("Error reading file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }
}

  .
  .
  .
```

6.23 sfs_fresize

6.23.1 Description

The **sfs_fresize** function is used to resize an opened file. Maximum size that can be specified for this function is 2^{31}-1.

The function sends an **SDD_FILE_RESIZE** message to the controller process of a file object and waits for an **SDD_FILE_RESIZE_REPLY** message. The calling process will be blocked until this message is received.

If the function succeeds, end-of-file indicator is reset. Refer to 3.8.12 for details about end-of-file indicator.

Following rules apply when using **sfs_fresize** function:

- If new **size** is equal to file size, the state of a file object does not change
- If new **size** is greater than current file size, the file is expanded and current file position pointer is set to the end of file.
- If new **size** is less than current file size, the file is truncated to the new size. If current position pointer is located somewhere after the new size, it will be moved to the end of the truncated file. Otherwise, the current position pointer is left unchanged.

6.23.2 Syntax

```
int sfs_fresize (
   sdd_obj_t * file,
   size_t size
);
```

6.23.3 Parameter

file	File descriptor.
	Specifies an opened file that shall be resized.
size	New file size in range 0..2GB-1

6.23.4 Return Value

If the function succeeds the return value is 0.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.23.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_FILE_RESIZE** (5.8.2.5).

error code	Return value of **sc_miscErrnoGet**

	After an **sfs_fresize** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate a message.
ERANGE	Size exceeds data type range (>2GB-1).

6.23.6 Source Code

The source code of the **sfs_fresize** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fresize.c

6.23.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_fresize(file, 1024) < 0) {
    PRINTF ("Error resizing file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.24 sfs_fresize64

6.24.1 Description

The **sfs_fresize64** function is used to resize an opened file. Maximum size that can be specified for this function is 2^{32}-1.

The function sends an **SDD_FILE_RESIZE64** message to the controller process of a file object and waits for an **SDD_FILE_RESIZE64_REPLY** message. The calling process will be blocked until this message is received.

If the function succeeds, end-of-file indicator is reset. Refer to 3.8.12 for details about end-of-file indicator.

Following rules apply when using **sfs_fresize64** function:

- If new **size** is equal to file size, the state of a file object does not change
- If new **size** is greater than current file size, the file is expanded and current file position pointer is set to the end of file.
- If new **size** is less than current file size, the file is truncated to the new size. If current position pointer is located somewhere after the new size, it will be moved to the end of the truncated file. Otherwise, the current position pointer is left unchanged.

6.24.2 Syntax

```
int sfs_fresize64 (
   sdd_obj_t * file,
   uint64_t size
);
```

6.24.3 Parameter

file	File descriptor.
	Specifies an opened file that shall be resized.
size	New file size in range 0..4GB-1

6.24.4 Return Value

If the function succeeds the return value is 0.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.24.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_FILE_RESIZE64** (5.9.2.5).

error code	Return value of **sc_miscErrnoGet**

	After an **sfs_fresize64** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate a message.

6.24.6 Source Code

The source code of the **sfs_fresize64** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fresize64.c

6.24.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_fresize64(file, 1024) < 0) {
  PRINTF ("Error resizing file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.25 sfs_fseek

6.25.1 Description

The **sfs_fseek** function is used to change current read/write offset of an opened disk/partition/file.

The function sends an **SDD_FILE_SEEK** message to the controller process of a disk/partition/file object and waits for an **SDD_FILE_SEEK_REPLY** message. The calling process will be blocked until this message is received.

If the function succeeds, end-of-file indicator is reset. Refer to 3.8.12 for details about end-of-file indicator.

Following rules apply when using **sfs_fseek** function on FAT file objects:

- If **whence** is set to **SEEK_SET** (file beginning is an origin), offset may be in range (0)..(2^32-1), inclusive, but cannot go beyond file size.
- If **whence** is set to **SEEK_CUR** (current file position pointer is an origin), offset may be positive, negative or zero, and is added to the current file position pointer. Result of adding must fall in range (0)..(filesize), inclusive.
- If **whence** is set to **SEEK_END** (file end is an origin), offset may be negative or zero and is added to the file size. Result of adding must fall in range (0)..(filesize), inclusive.

Following rules apply when using **sfs_fseek** function on physical drive or partition objects opened for raw access (refer to 3.7.12 for details of raw access):

- If **whence** is set to **SEEK_SET** (drive/partition beginning is an origin), offset may be in range (0)..(2^32-sectorsize), inclusive, but cannot go beyond drive/partition size.
- If **whence** is set to **SEEK_CUR** (current drive/partition position pointer is an origin), offset may be positive, negative or zero, and is added to the current drive/partition position pointer. Result of adding must fall in range (0)..(size), inclusive.
- If **whence** is set to **SEEK_END** (drive/partition end is an origin), offset may be negative or zero and is added to the drive/partition size. Result of adding must fall in range (0)..(size), inclusive.

6.25.2 Syntax

```
int sfs_fseek (
   sdd_obj_t * file,
   off_t off,
   flags_t whence
);
```

6.25.3 Parameter

file	Disk/partition/file descriptor.
	Specifies a disk/partition/file to seek.
off	Seek offset.
	Specifies a seek offset in bytes, relative to the origin.
whence	Offset origin.
	Specifies an origin from which a new position shall be calculated. Valid values are SEEK_SET, SEEK_CUR and SEEK_END.

6.25.4 Return Value

If the function succeeds the return value is 0.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.25.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_FILE_SEEK** (5.10.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fseek** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.25.6 Source Code

The source code of the **sfs_fseek** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fseek.c

6.25.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_fseek(file, 1024, SEEK_SET) < 0) {
  PRINTF ("Error seeking file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.26 sfs_fseek64

6.26.1 Description

The **sfs_fseek64** function is used to change current read/write offset of an opened disk/partition/file.

The function sends an **SDD_FILE_SEEK64** message to the controller process of a disk/partition/file object and waits for an **SDD_FILE_SEEK64_REPLY** message. The calling process will be blocked until this message is received.

If the function succeeds, end-of-file indicator is reset. Refer to 3.8.12 for details about end-of-file indicator.

Following rules apply when using **sfs_fseek64** function on FAT file objects:

- If **whence** is set to **SEEK_SET** (file beginning is an origin), offset may be in range (0)..(2^32-1), inclusive, but cannot go beyond file size.
- If **whence** is set to **SEEK_CUR** (current file position pointer is an origin), offset may be positive, negative or zero, and is added to the current file position pointer. Result of adding must fall in range (0)..(filesize), inclusive.
- If **whence** is set to **SEEK_END** (file end is an origin), offset may be negative or zero and is added to the file size. Result of adding must fall in range (0)..(filesize), inclusive.

Following rules apply when using **sfs_fseek64** function on physical drive or partition objects opened for raw access (refer to 3.7.12 for details of raw access):

- If **whence** is set to **SEEK_SET** (drive/partition beginning is an origin), offset may be in range (0)..(2^64-sectorsize), inclusive, but cannot go beyond drive/partition size.
- If **whence** is set to **SEEK_CUR** (current drive/partition position pointer is an origin), offset may be positive, negative or zero, and is added to the current drive/partition position pointer. Result of adding must fall in range (0)..(size), inclusive.
- If **whence** is set to **SEEK_END** (drive/partition end is an origin), offset may be negative or zero and is added to the drive/partition size. Result of adding must fall in range (0)..(size), inclusive.

6.26.2 Syntax

```
int sfs_fseek64 (
   sdd_obj_t * file,
   int64_t off,
   flags_t whence
);
```

6.26.3 Parameter

file	Disk/partition/file descriptor.
	Specifies a disk/partition/file to seek.
off	Seek offset.
	Specifies a seek offset in bytes, relative to the origin.
whence	Offset origin.
	Specifies an origin from which a new position shall be calculated. Valid values are SEEK_SET, SEEK_CUR and SEEK_END.

6.26.4 Return Value

If the function succeeds the return value is 0.

If the function fails the return value is -1 and file error indicator is set.

To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.26.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_FILE_SEEK64** (5.11.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fseek64** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.26.6 Source Code

The source code of the **sfs_fseek64** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fseek64.c

6.26.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "r");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_fseek64(file, 1024, SEEK_SET) < 0) {
  PRINTF ("Error seeking file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.27 sfs_ftell

6.27.1 Description

The **sfs_ftell** function is used to get a current file position pointer.

The function sends an **SFS_FTELL** message to the controller process of a file object (or physical drive or a partition accessed raw) and waits for an **SFS_FTELL_REPLY** message. The calling process will be blocked until this message is received.

6.27.2 Syntax

```
int sfs_ftell (
   sdd_obj_t * object
);
```

6.27.3 Parameter

object	File object descriptor.
	Specifies a file object for a position pointer to be returned.

6.27.4 Return Value

If the function succeeds the return value is a file pointer position.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.27.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FTELL** (5.36.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_ftell** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate a message.
ERANGE	Position returned exceeds return data type range.

6.27.6 Source Code

The source code of the **sfs_ftell** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_ftell.c

6.27.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[10] = {"test text"};
size_t characters_written;
int position;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

characters_written = sfs_fwrite(buffer, 1, sizeof(buffer), file);

if (characters_written != sizeof(buffer)) {
  PRINTF ("Error writing file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

position = sfs_ftell(file);
if (position < 0) {
  PRINTF ("Error getting file pointer position\n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.28 sfs_ftell64

6.28.1 Description

The **sfs_ftell64** function is used to get a current file position pointer.

The function sends an **SFS_FTELL64** message to the controller process of a file object (or physical drive or a partition accessed raw) and waits for an **SFS_FTELL64_REPLY** message. The calling process will be blocked until this message is received.

6.28.2 Syntax

```
int64_t sfs_ftell64 (
   sdd_obj_t * object
);
```

6.28.3 Parameter

object	File object descriptor.
	Specifies a file object for a position pointer to be returned.

6.28.4 Return Value

If the function succeeds the return value is current file pointer position.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.28.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_FTELL64** (5.37.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_ftell64** call.
EFSNULLPOINTER	File descriptor pointer is NULL.
ENOMEM	No memory to allocate a message.
ERANGE	Position returned exceeds return data type range.

6.28.6 Source Code

The source code of the **sfs_ftell64** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_ftell64.c

6.28.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
char buffer[10] = {"test text"};
size_t characters_written;
int64_t position;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

characters_written = sfs_fwrite(buffer, 1, sizeof(buffer), file);

if (characters_written != sizeof(buffer)) {
  PRINTF ("Error writing file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

position = sfs_ftell64(file);

if (position < 0) {
  PRINTF ("Error getting file pointer position\n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.29 sfs_fwrite

6.29.1 Description

The **sfs_fwrite** function is used to write data to an opened disk/partition/file.

The function sends an **SDD_DEV_WRITE** message to the writer process of a disk/partition/file object and waits for an **SDD_DEV_WRITE_REPLY** message. The calling process will be blocked until this message is received.

6.29.2 Syntax

```
size_t sfs_fwrite (
   const void * buf,
   size_t size,
   size_t count,
   sdd_obj_t * file
);
```

6.29.3 Parameter

buf	Buffer.
	Specifies a buffer containing the data to write to the disk/partition/file.
size	Size of each element to write.
	Specifies size in bytes of each elements that will be written to file.
count	Number of elements to write.
	Specifies number of elements to write, each of size of **size** bytes.
file	Descriptor of a disk/partition/file.
	Specifies a disk/partition/file to write to.

6.29.4 Return Value

Function returns number of elements successfully written to a disk/partition/file.

If either **size** or **count** parameter is zero, the function returns zero and state of file does not change.

If return value is different than parameter **count**, write error occurred. To check if error indicator has been set, use function **sfs_ferror** (6.12). Refer to 3.8.12 for details about file error indicator.

If any error occurs in the function call, process errno variable is also set to the same value as file error indicator. Process errno variable can be retrieved by using **sc_miscErrnoGet**.

6.29.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_DEV_WRITE** (5.7.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_fwrite** call.
EFSNULLPOINTER	File descriptor pointer is NULL or buffer pointer is NULL.

ERANGE	Number of bytes to read exceeds parameter type range.

6.29.6 Source Code

The source code of the **sfs_fwrite** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_fwrite.c

6.29.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;
  char buffer[10] = {"test text"};
  size_t characters_written;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_fopen(chanFs, "/ramdisk0/p2/testdir/testfile.txt", "w");
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  characters_written = sfs_fwrite(buffer, 1, sizeof(buffer), file);

  if (characters_written != sizeof(buffer)) {
    PRINTF ("Error writing file \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.30 sfs_get

6.30.1 Description

The **sfs_get** function is used to retrieve filesystem object: physical drive, partition, directory or a file.

The function sends an **SDD_MAN_GET** message to the controller process of a filesystem object and waits for an **SDD_MAN_GET_REPLY** message. The calling process will be blocked until this message is received.

If **dir** reference object is **not** specified, an absolute path must be used. For the format of absolute paths refer to 3.4.1.

If **dir** reference object is specified, a relative path must be used. For the format of relative paths refer to 3.4.2.

6.30.2 Syntax

```
sdd_obj_t * sfs_get (
   sdd_obj_t * dir,
   const char * name,
   sc_msgid_t type
);
```

6.30.3 Parameter

dir	Descriptor of a starting point object.
	Specifies an object to start the search from.
name	Path of an object to get.
type	Type of an object to get.
	Specifies a type of an object to get. **SFS_ATTR_ANY** means any type, else given type must match exactly. Other possible values are: • **SFS_ATTR_FILE** - FAT file or raw access to physical drive or a partition, • **SFS_ATTR_DIR** - FAT directory, • **SFS_ATTR_PARTITION** - partition, • **SFS_ATTR_DRIVE** - physical drive, • **SFS_ATTR_ROOT** – filesystem root.

6.30.4 Return Value

If the function succeeds the return value is an object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.30.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_DUP** (5.19.3.3, 5.19.4.3) and **SDD_MAN_GET** (5.13.3.3, 5.13.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_get** call.
EBADF	Relative path must have a directory.

EFSEMPTYNAME	Name cannot be empty.
EFSINVALNAME	Name is invalid.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted any-more.
EFSNOROOT	Parameter 'dir' is NULL and device manager process not found.
EFSNULLPOINTER	Name is NULL.
ENAMETOOLONG	Name is too long.
ENOENT	Requested object not found.

6.30.6 Source Code

The source code of the **sfs_get** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_get.c

6.30.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * disk;
sdd_obj_t * partition;
sdd_obj_t * directory;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* get physical disk object */
disk = sfs_get(chanFs, "ramdisk0", SFS_ATTR_DRIVE);
if (disk == NULL) {
  PRINTF ("Could not get disk object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* get partition object */
partition = sfs_get(disk, "p2", SFS_ATTR_PARTITION);
if (disk == NULL) {
  PRINTF ("Could not get disk object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* get directory object */
directory = sfs_get(partition, "testdir", SFS_ATTR_DIR);
if (disk == NULL) {
  PRINTF ("Could not get disk object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* get file object */
file = sfs_get(directory, "testfile.txt", SFS_ATTR_FILE);
```

```
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

      .
      .
      .
```

6.31 sfs_getcwd

6.31.1 Description

The **sfs_getcwd** function is used to get a current working directory of a directory or a file object.

The function sends multiple **SDD_MAN_GET** messages to the controller process of a filesystem object and waits for **SDD_MAN_GET_REPLY** messages. The calling process will be blocked until these messages are received.

This function returns a path, which is relative to filesystem root.

6.31.2 Syntax

```
int sfs_getcwd (
   sdd_obj_t * object,
   char * path,
   size_t len
);
```

6.31.3 Parameter

object	Descriptor of an object.
	Specifies a directory to get a current path of.
path	Buffer to store the path to.
len	Size of path buffer.

6.31.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.31.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_get** (6.30.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_getcwd** call.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted any-more.
EFSNOTFSOBJECT	Not a file system object.
EFSNULLPOINTER	Parameter 'object' or 'path' is NULL.
ENAMETOOLONG	Current working directory path is too long to fit into 'path'.

6.31.6 Source Code

The source code of the **sfs_getcwd** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_getcwd.c

6.31.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * directory;
char path[128];

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

directory = sfs_get(chanFs, "/ramdisk0/p2/testdir", SFS_ATTR_DIR);
if (directory == NULL) {
  PRINTF ("Could not get directory object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_getcwd(directory, path, sizeof(path)) < 0) {
  PRINTF ("Could not get directory path \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* path contains "/ramdisk0/p2/testdir" */

    .
    .
    .
```

6.32 sfs_getFirst

6.32.1 Description

The **sfs_getFirst** function is used to get first physical drive object, first mounted partition, first directory or first file from current working directory of a filesystem object.

The function sends an **SDD_MAN_GET_FIRST** message to the controller process of a filesystem object and waits for an **SDD_MAN_GET_FIRST_REPLY** message. The calling process will be blocked until this message is received.

6.32.2 Syntax

```
sdd_obj_t *sfs_getFirst (
   sdd_obj_t * dir
);
```

6.32.3 Parameter

dir	Directory descriptor (filesystem root, physical drive, partition, or a FAT directory).
	Specifies a directory from which to get first object (drive, partition, directory or file)

6.32.4 Return Value

If the function succeeds the return value is retrieved first object.

If the function fails or no objects in a directory, the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.32.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_getFirst** call. For all errors please refer to **SDD_MAN_GET_FIRST** (5.14.3.3, 5.14.4.3).

6.32.6 Source Code

The source code of the **sfs_getFirst** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_get.c

6.32.7 Example

```
    .
    .
    .

  logd_t *logd;
  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * directory;
  sdd_obj_t * item;
  sdd_obj_t * previous;

  logd = logd_new ("/SCP_logd",
                   LOGD_INFO,
                   "test",
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* get directory object */
  directory = sfs_get(chanFs, "/ramdisk0/p2/testdir", SFS_ATTR_DIR);
  if (directory == NULL) {
    PRINTF ("Could not get directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  item = sfs_getFirst(directory);
  while (item) {
    logd_printf(logd, LOGD_INFO, "Item: %s \n", item->name);

    previous = item;
    item = sfs_getNext(directory, previous);
    if (sfs_free(&previous)) {
      PRINTF ("Could not free object \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
  }

    .
    .
    .
```

6.33 sfs_getNext

6.33.1 Description

The **sfs_getNext** function is used to get next physical drive object, first mounted partition, first directory or first file after a specified object.

The function sends an **SDD_MAN_GET_NEXT** message to the controller process of a filesystem object and waits for an **SDD_MAN_GET_NEXT_REPLY** message. The calling process will be blocked until this message is received.

6.33.2 Syntax

```
sdd_obj_t *sfs_getNext (
   sdd_obj_t * dir,
   sdd_obj_t * previous
);
```

6.33.3 Parameter

dir	Directory descriptor.
	Specifies a directory from which to get next object (drive, partition, directory or file)
previous	Previous object descriptor.
	Specifies a previously fetched object.

6.33.4 Return Value

If the function succeeds the return value is retrieved next object.

If the function fails or no more objects to return, the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.33.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_getNext** call.
	For all errors please refer to **SDD_MAN_GET_NEXT** (5.15.3.3, 5.15.4.3).

6.33.6 Source Code

The source code of the **sfs_getNext** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_get.c

6.33.7 Example

```
    .
    .
    .

  logd_t *logd;
  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * directory;
  sdd_obj_t * item;
  sdd_obj_t * previous;

  logd = logd_new ("/SCP_logd",
                   LOGD_INFO,
                   "test",
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* get directory object */
  directory = sfs_get(chanFs, "/ramdisk0/p2/testdir", SFS_ATTR_DIR);
  if (directory == NULL) {
    PRINTF ("Could not get directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  item = sfs_getFirst(directory);
  while (item) {
    logd_printf(logd, LOGD_INFO, "Item: %s \n", item->name);

    previous = item;
    item = sfs_getNext(directory, previous);
    if (sfs_free(&previous)) {
      PRINTF ("Could not free object \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
  }

    .
    .
    .
```

6.34 sfs_getProperty

6.34.1 Description

The **sfs_getProperty** function is used to get a specific property of a filesystem object.

The function sends an **SDD_OBJ_TAG_GET** message to the controller process of a filesystem object and waits for an **SDD_OBJ_TAG_GET_REPLY** message. The calling process will be blocked until this message is received.

6.34.2 Syntax

```
int sfs_getProperty (
   sdd_obj_t *obj,
   sc_tag_t tag,
   void *data,
   size_t size
);
```

6.34.3 Parameter

obj	Descriptor of an object.
	Specifies an object to get the property of.
tag	Property name.
	Specifies a name of a property to get.
data	Buffer store retrieved property data.
size	Property data size.
	Specifies a maximum size **data** can hold.

6.34.4 Supported properties types

FAT filesystem supports getting the following properties:

- Partition label
- Partition serial number
- Directory/file attributes
- File cache size

6.34.4.1 Partition label

This property type is a FAT partition label. Following conditions must be met to correctly retrieve the property value.

u.ioctl.tag:	`CHANFS_PROPERTY_PARTITION_LABEL`
Data type of **u.ioctl.data**:	Character array
Size of **u.ioctl.data:**	`CHANFS_PROPERTY_SIZE_PARTITION_LABEL`

6.34.4.2 Partition serial number

This property type is a FAT partition serial number. Following conditions must be met to correctly retrieve the property value.

u.ioctl.tag:	`CHANFS_PROPERTY_PARTITION_SERIAL_NUMBER`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of u.ioctl.data:	`CHANFS_PROPERTY_SIZE_PARTITION_SERIAL_NUMBER`

6.34.4.3 Directory/file attributes

This property type is a directory/file attributes. Following conditions must be met to correctly retrieve the property value.

u.ioctl.tag:	`CHANFS_PROPERTY_FILE_DIR_ATTRIBUTES`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of u.ioctl.data:	`CHANFS_PROPERTY_SIZE_FILE_DIR_ATTRIBUTES`

Attributes are encoded using following format:

Bit <3>	Directory/file is **hidden**.
Bit <2>	Directory/file is **archived**.
Bit <1>	Directory/file is **system type**.
Bit <0>	Directory/file is **read-only**.

A set of macro masks is provided to decode each attribute from retrieved property value:

`CHANFS_ATTRIBUTE_READONLY_GET(data)` – Get read-only attribute.

`CHANFS_ATTRIBUTE_ARCHIVE_GET(data)` – Get archive attribute.

`CHANFS_ATTRIBUTE_SYSTEM_GET(data)` – Get system attribute.

`CHANFS_ATTRIBUTE_HIDDEN_GET(data)` – Get hidden attribute.

6.34.4.4 File cache size

This property type is a file cache size in units of sectors.

u.ioctl.tag:	`CHANFS_PROPERTY_FILE_CACHE_SIZE`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of u.ioctl.data:	`CHANFS_PROPERTY_SIZE_FILE_CACHE_SIZE`

6.34.4.5 List of allocated objects

This property type is a debug feature which allows to retrieve a list of currently allocated object for a specific physical drive drive.

This property uses structure **chanfs_allocated_objects_t** and stores paths of allocated objects in a form of character strings separated by NULL characters in the structure member **allocated_objects**. Number of paths is returned in the structure member **objects_count**.

If allocated size of this structure is not sufficient to hold all the paths, the member **required_memory** will hold the required size of **chanfs_allocated_objects_t** structure to fit all the paths. Otherwise, if structure size is sufficient, the member **required_memory** will hold value of 0.

u.ioctl.tag:	`CHANFS_PROPERTY_ALLOCATED_OBJECTS`
Data type of **u.ioctl.data**:	`chanfs_allocated_objects_t`
Size of u.ioctl.data:	`CHANFS_PROPERTY_SIZE_ALLOCATED_OBJECTS +` size required for all allocated objects paths

Format of **chanfs_allocated_objects_t** structure:

```
typedef struct chanfs_allocated_objects_s {
  uint32_t required_memory;
  uint32_t objects_count;
  char allocated_objects[1];
} chanfs_allocated_objects_t;
```

6.34.4.6 Partition FAT cache size

This property type is a FAT cache size in units of sectors.

u.ioctl.tag:	`CHANFS_PROPERTY_FAT_CACHE_SIZE`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of u.ioctl.data:	`CHANFS_PROPERTY_SIZE_FAT_CACHE_SIZE`

6.34.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.34.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_TAG_GET** (5.23.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_getProperty** call.
EFSNULLPOINTER	Parameter 'obj' or 'data' is NULL.
ENOMEM	No memory to allocate a request.

6.34.7 Source Code

The source code of the **sfs_getProperty** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_property.c

6.34.8 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  sdd_obj_t * file;
  char label[CHANFS_PROPERTY_SIZE_PARTITION_LABEL];
  uint32_t serial_number;
  uint32_t attributes;
  int readonly, archive, system, hidden;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_getProperty(partition, CHANFS_PROPERTY_PARTITION_LABEL, label,
                                                       sizeof(label)) < 0) {
    PRINTF ("Could not get partition label \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_getProperty(partition, CHANFS_PROPERTY_PARTITION_SERIAL_NUMBER, &serial_number,
                                               sizeof(serial_number)) < 0) {
    PRINTF ("Could not get partition serial number \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_getProperty(file, CHANFS_PROPERTY_FILE_DIR_ATTRIBUTES, &attributes,
                                                  sizeof(attributes)) < 0) {
    PRINTF ("Could not get file attributes \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  readonly = CHANFS_ATTRIBUTE_READONLY_GET(attributes);
  archive  = CHANFS_ATTRIBUTE_ARCHIVE_GET(attributes);
  system   = CHANFS_ATTRIBUTE_SYSTEM_GET(attributes);
  hidden   = CHANFS_ATTRIBUTE_HIDDEN_GET(attributes);

    .
    .
    .
```

6.35 sfs_getShortName

6.35.1 Description

The **sfs_getShortName** function is used to get a short name (DOS 8.3 format) of a directory or a file.

The function sends an **SFS_GETSHORTNAME** message to the controller process of a filesystem object and waits for an **SFS_GETSHORTNAME_REPLY** message. The calling process will be blocked until this message is received.

6.35.2 Syntax

```
int sfs_getShortName(
   sdd_obj_t * obj,
   char *buf,
   size_t size
);
```

6.35.3 Parameter

obj	Object descriptor.
	Specifies an object (directory or file) to get a shortname of.
buf	Buffer.
	Specifies a buffer to store a short name to.
size	Buffer size.
	Specifies a size of the buffer.

6.35.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.35.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_GETSHORTNAME** (5.38.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_getShortName** call.
ENOMEM	No memory to allocate message.

6.35.6 Source Code

The source code of the **sfs_getShortName** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_get.c

6.35.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * directory;
  char name[64];

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  directory = sfs_get(chanFs, "/ramdisk0/p2/testdirectory", SFS_ATTR_DIR);
  if (directory == NULL) {
    PRINTF ("Could not get directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_getShortName(directory, name, sizeof(name)) < 0) {
    PRINTF ("Could not get short name of the directory object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* short name returned: "TESTDI~1" */

    .
    .
    .
```

6.36 sfs_ioctl

6.36.1 Description

The **sfs_ioctl** function is used to get or set certain properties of a filesystem object.

The function sends an **SDD_DEV_IOCTL** message to the controller process of a filesystem object and waits for an **SDD_DEV_IOCTL_REPLY** message. The calling process will be blocked until this message is received.

6.36.2 Syntax

```
int sfs_ioctl (
   const sdd_obj_t *obj,
   unsigned int cmd,
   unsigned long arg
);
```

6.36.3 Parameter

obj	Descriptor of an object.
	Specifies an object to get or set the property of.
cmd	Object specific command.
	Specifies an ioctl command to be executed on the object.
arg	Command specific argument.
	Command specific argument (may be a pointer).

6.36.4 Supported commands

Filesystem supports following commands:

- Drive size
- Partition size
- Filesystem mounted safe status (only SAFE FAT product)
- Init SAFE FAT filesystem (only SAFE FAT product)

6.36.4.1 Drive or partition size

This command is used to get a drive or a partition size, encoded in **blkdev_geometry_t** or in **blkdev_geometry64_t** structure. Drive or partition object must be opened in raw mode (refer to 3.7.12)

Small drives (< 2GB):

cmd:	`BLKDEVGETPRM`
Data type of **arg**:	Pointer to **blkdev_geometry_t** structure.

Large drives (>= 2GB):

cmd:	`BLKDEVGETPRM64`

Data type of **arg**:	Pointer to **blkdev_geometry64_t** structure.

6.36.4.2 Filesystem mounted safe status

This command is used to get a safe status of mounted filesystem.

cmd:	`CHANFS_IOCTL_CMD_IS_MOUNTED_SAFE`
Data type of **arg**:	Pointer to 32-bit unsigned integer.

If returned value is non-zero, filesystem is mounted in SAFE mode. Otherwise it is mounted non-SAFE.

6.36.4.3 Init SAFE FAT filesystem

This command is used to initialize SAFE FAT structures on a mounted, non-SAFE FAT filesystem partition/drive. The command will fail if root directory of a partition/drive contains entries named:

`__SFAT__`

or

`%%SFAT%%`

These entries must be deleted prior to using this ioctl command.

Refer to 4.4 for informations about initializing SAFE FAT structures.

cmd:	`CHANFS_IOCTL_CMD_INIT_SAFE_FAT`
arg:	Not used

6.36.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.36.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_DEV_IOCTL** (5.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_ioctl** call.
EFSNULLPOINTER	Parameter 'obj' is NULL.
ENOMEM	No memory to allocate a request.

6.36.7 Source Code

The source code of the **sfs_ioctl** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_ioctl.c

6.36.8 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * drive;
chanfs_mount_t mount;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

drive = sfs_get(chanFs, "/sdcard0", SFS_ATTR_DRIVE);
if (drive == NULL) {
  PRINTF ("Could not get physical drive object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* Mount entire drive */
mount.partition = 0;
mount.fatCacheSize = CHANFS_MOUNT_USE_DEFAULT_FAT_CACHE_SIZE;
mount.nonSafeKey = CHANFS_MOUNT_NON_SAFE_KEY;
if (sfs_mount(drive, &mount, sizeof(mount)) < 0) {
  PRINTF ("Error mounting partition \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_chdir(drive, “p1”) < 0) {
  PRINTF ("Could not change to partitition \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_ioctl(drive, CHANFS_IOCTL_CMD_INIT_SAFE_FAT, 0) < 0) {
  PRINTF ("Error creating SAFE FAT structures \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.37 sfs_mount

6.37.1 Description

The **sfs_mount** function is used to mount a physical drive or a partition to make it available for the user.

The function sends an **SFS_MOUNT** message to the controller process of a physical drive object and waits for an **SFS_MOUNT_REPLY** message. The calling process will be blocked until this message is received.

6.37.2 Syntax

```
int sfs_mount (
   sdd_obj_t * drive,
   void *custom,
   size_t custom_size
);
```

6.37.3 Parameter

drive	Drive descriptor.
	Specifies a drive descriptor. A partition or paritions on this drive will be mounted.
custom	Mounting parameters (filesystem dependent).
	Specifies which partition shall be mounted.
custom_size	Size of mounting parameters (filesystem dependent).

6.37.4 Mounting parameters

For FAT filesystem the mounting parameters are defined as a structure:

```
typedef struct chanfs_mount_s {
  int partition;
  uint32_t fatCacheSize;
} chanfs_mount_t;
```

For SAFE FAT filesystem the mounting parameters are defined as a structure:

```
typedef struct chanfs_mount_s {
  int partition;
  uint32_t fatCacheSize;
  int nonSafeKey;
} chanfs_mount_t;
```

where:

partition	Partition number to mount.
0	Mount an entire drive. This value can be used in case a physical drive contains a single filesystem.
1-4	Mount a specified partition. This value can be used in case a physical drive is partitioned.
-1	Try to mount all filesystems available on a physical drive.
fatCacheSize	FAT cache size used for this filesystem expressed in number of sectors.

N	FAT cache size is N sectors, or specify CHANFS_MOUNT_USE_DEFAULT_FAT_CACHE_SIZE to use default FAT cache size from configuration file (3.13.6.5.1)
nonSafeKey	Non safe mount key.
	If this member is set to CHANFS_MOUNT_NON_SAFE_KEY, mount will always be in a non-SAFE mode, even if there are valid SAFE FAT structures.

6.37.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.37.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_MOUNT** (5.39.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_mount** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.37.7 Source Code

The source code of the **sfs_mount** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_mount.c

6.37.8 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * drive;
chanfs_mount_t mount;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
if (drive == NULL) {
  PRINTF ("Could not get physical drive object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* Mount 2nd partition on drive */

mount.partition = 2;
mount.fatCacheSize = CHANFS_MOUNT_USE_DEFAULT_FAT_CACHE_SIZE;
if (sfs_mount(drive, &mount, sizeof(mount)) < 0) {
  PRINTF ("Error mounting partition \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.38 sfs_move

6.38.1 Description

The **sfs_move** function is used to rename a directory/file and/or move it to another location within the same partition.

The function sends an **SFS_MOVE** message to the controller process of an object to be renamed and/or moved and waits for an **SFS_MOVE_REPLY** message. The calling process will be blocked until this message is received.

6.38.2 Syntax

```
int sfs_move (
   sdd_obj_t ** object,
   sdd_obj_t * newman,
   const char * newname
);
```

6.38.3 Parameter

object	Object descriptor.
	Specifies an object which shall be moved and/or name shall be changed.
newman	New directory to move an object to.
	Specifies a new directory where object shall be moved. Leaving this parameter NULL keeps the file in the same directory.
newname	New object name.
	Specifies a new object name. Leaving this parameter NULL keeps the original object's name.

6.38.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.38.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_MOVE** (5.40.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_move** call.
EFSNOTSAMEPART	Source and destination are not on the same partition.
EFSNULLPOINTER	Parameter 'object' is NULL or 'object' points to a NULL pointer or 'newman' and 'newname' cannot be both NULL.
ENOMEM	Out of memory.

6.38.6 Source Code

The source code of the **sfs_move** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_move.c

6.38.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;
sdd_obj_t * newdir;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

newdir = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (newdir == NULL) {
  PRINTF ("Could not get newdir object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_move(&file, newdir, "testfile2.txt") < 0) {
  PRINTF ("Could not move and rename file \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_get(chanFs, "/ramdisk0/p2/testfile2.txt", SFS_ATTR_FILE);
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.39 sfs_move2

6.39.1 Description

The **sfs_move2** function is used to copy a directory/file to another location (and rename, if required) within the same partition or between different partitions and disks.

The function sends multiple messages to the controller process of a filesystem object and waits for reply messages. The calling process will be blocked until these messages are received.

The **sfs_move2** function uses only paths, which are relative to **root** object. For the format of relative paths, refer to 3.4.2.

When using **sfs_move2**, following rules apply:

- If **oldpath** specifies a directory and **newpath** specifies a directory which does not exist: **sfs_move2** first creates the new directory and copies **only the content of the old directory** to the newly created one. When all the data is copied successfully, the **sfs_move2** recursively deletes an old directory with its content.
- If **oldpath** specifies a directory and **newpath** specifies a directory which already exists: **sfs_move2** copies an **old directory including its content** to the new directory. When all the data is copied successfully, the **sfs_move2** recursively deletes an old directory with its content.

The **sfs_move2** is a function which uses recursion. The maximum depth of recursion is specified by macro define **SFS_DIR_CPY_MAX_DEPTH** in source code file of this function (refer to 6.5.6 for location of the source code). Default maximum depth level is set to 10. If different level is required, change the macro definition and rebuild Sciopta File System library.

6.39.2 Syntax

```
int sfs_move2 (
   sdd_obj_t * root,
   const char * oldpath,
   const char * newpath
);
```

6.39.3 Parameter

root	Filesystem object descriptor.
	Specifies a filesystem directory object.
oldpath	Old path.
	Specifies a path to the object which shall be moved.
newpath	New path.
	Specifies a new path for the object.

6.39.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.39.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **sfs_getcwd** (6.31.5), **sfs_create** (6.6.5), **sfs_get** (6.30.5), **SDD_DEV_OPEN** (5.5.2.5), **SDD_DEV_READ** (5.6.2.5), **SDD_DEV_WRITE** (5.7.2.5), **SDD_MAN_GET** (5.13.3.3, 5.13.4.3) and **SDD_MAN_RM** (5.18.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_move2** call.
EEXIST	Paths are the same or file already exists.
EFSEMPTYNAME	Path cannot be empty.
EFSINVALNAME	Path is invalid.
EFSLIBINTERNAL	Internal error detected inside library functions. Filesystem cannot be trusted any-more.
EFSNULLPOINTER	Parameter 'root' is NULL or path is NULL.
EFSTOODEEP	Maximum objects depth reached.
EINVAL	Not-last segment of path is a file.
ENAMETOOLONG	New path is too long or old path is too long or recursive path is too long.
ENOENT	Old path not found or new path not found.

6.39.6 Source Code

The source code of the **sfs_move2** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_move.c

6.39.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * directory;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_move2(chanFs, "/ramdisk0/p2/testdir", "/ramdisk0/p1") < 0) {
  PRINTF ("Could not move directory and its content \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

directory = sfs_get(chanFs, "/ramdisk0/p4/testdir", SFS_ATTR_DIR);
if (directory == NULL) {
  PRINTF ("Could not get directory object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}
```

.
.
.

6.40 sfs_objType

6.40.1 Description

The **sfs_objType** is a macro which is used to get type of an object.

6.40.2 Syntax

```
#define sfs_objType(obj)   ((obj)->type)
```

6.40.3 Parameter

object	Object descriptor.
	Specifies an object which shall be moved and/or name shall be changed.
newman	New directory to move an object to.
	Specifies a new directory where object shall be moved. Leaving this parameter NULL keeps the file in the same directory.
newname	New object name.
	Specifies a new object name. Leaving this parameter NULL keeps the original object's name.

6.40.4 Return Value

The macro return an object type. Valid object types are:

File object	SFS_ATTR_FILE
Directory object	SFS_ATTR_DIR
Partition object	SFS_ATTR_PARTITION
Drive object	SFS_ATTR_DRIVE
Filesystem root object	SFS_ATTR_ROOT

6.40.5 Errors

This macro does not return any errors.

6.40.6 Source Code

The source code of the **sfs_objType** macro can be found here:

<installation_folder>\sciopta\<version>\include\sfs\sfs.h

6.41 sfs_open

6.41.1 Description

The **sfs_open** function is used to open a disk/partition/file object for read, write or read/write.

The function sends an **SDD_DEV_OPEN** message to the controller process of a disk/partition/file object and waits for an **SDD_DEV_OPEN_REPLY** message. The calling process will be blocked until this message is received.

The **sfs_open** function uses an already present file object to open a file. To open a file by using path, **sfs_fopen** (6.41) may be used.

6.41.2 Syntax

```
sdd_obj_t * sfs_open (
   sdd_obj_t * file,
   flags_t flags
);
```

6.41.3 Parameter

file	Specifies disk/partition/file object to open.
	Disk/partition/file to be opened.
flags	Open flags.
	Specifies flags to be used when opening the file. Refer to 5.5.2.4 for the list of valid flags.

6.41.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.41.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_DEV_OPEN** (5.5.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_open** call.
EFSNULLPOINTER	File descriptor pointer is NULL.

6.41.6 Source Code

The source code of the **sfs_open** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_open.c

6.41.7 Example

```
    .
    .
```

```
  .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * file;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
if (file == NULL) {
  PRINTF ("Could not get file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_open(file, O_RDWR) < 0) {
  PRINTF ("Could not open file object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

  .
  .
  .
```

6.42 sfs_setProperty

6.42.1 Description

The **sfs_setProperty** function is used to set a specific property of a filesystem object.

The function sends an **SDD_OBJ_TAG_SET** message to the controller process of a filesystem object and waits for an **SDD_OBJ_TAG_SET_REPLY** message. The calling process will be blocked until this message is received.

6.42.2 Syntax

```
int sfs_setProperty (
   sdd_obj_t *obj,
   sc_tag_t tag,
   const void *data,
   size_t size
);
```

6.42.3 Parameter

obj	Descriptor of an object.
	Specifies an object to set the property of.
tag	Property name.
	Specifies a name of a property to set.
data	Buffer containing property data.
size	Property data size.
	Specifies a size **data** holds.

6.42.4 Supported properties types

FAT filesystem supports setting the following properties:

- Partition label
- Directory/file attributes
- File cache size

6.42.4.1 Partition label

This property type is a FAT partition label. Following conditions must be met to correctly set the new property value.

u.ioctl.tag:	`CHANFS_PROPERTY_PARTITION_LABEL`
Data type of **u.ioctl.data**:	Character array
Size of **u.ioctl.data**:	`CHANFS_PROPERTY_SIZE_PARTITION_LABEL`
Value of **u.ioctl.data**:	New partitional label + null terminating character

6.42.4.2 Directory/file attributes

This property type is a FAT partition serial number. Following conditions must be met to correctly set the property value.

u.ioctl.tag:	`CHANFS_PROPERTY_FILE_DIR_ATTRIBUTES`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of **u.ioctl.data**:	`CHANFS_PROPERTY_SIZE_FILE_DIR_ATTRIBUTES`
Value of **u.ioctl.data**:	New directory/file attributes

New attributes are encoded using following format:

Bit <7>	If set then **hidden** attribute will be changed.
Bit <6>	If set then **system** attribute will be changed.
Bit <5>	If set then **archive** attribute will be changed.
Bit <4>	If set then **read-only** attribute will be changed.
Bit <3>	Directory/file is **hidden**.
Bit <2>	Directory/file is **archived**.
Bit <1>	Directory/file is **system** type.
Bit <0>	Directory/file is **read-only**.

A set of macro masks is provided to encode attributes desired to be changed. Macros must be OR'ed to get a desirable attributes change.

`CHANFS_ATTRIBUTE_READONLY_SET` – Set **read-only** attribute.

`CHANFS_ATTRIBUTE_READONLY_CLR` – Clear **read-only** attribute.

`CHANFS_ATTRIBUTE_ARCHIVE_SET` – Set **archive** attribute.

`CHANFS_ATTRIBUTE_ARCHIVE_CLR` – Clear **archive** attribute.

`CHANFS_ATTRIBUTE_SYSTEM_SET` – Set **system** attribute.

`CHANFS_ATTRIBUTE_SYSTEM_CLR` – Clear **system** attribute.

`CHANFS_ATTRIBUTE_HIDDEN_SET` – Set **hidden** attribute.

`CHANFS_ATTRIBUTE_HIDDEN_CLR` – Clear **hidden** attribute.

6.42.4.3 File cache size

This property type is a file cache size in units of sectors. Following conditions must be met to correctly set the property value.

u.ioctl.tag:	`CHANFS_PROPERTY_FILE_CACHE_SIZE`

Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of **u.ioctl.data**:	`CHANFS_PROPERTY_SIZE_FILE_CACHE_SIZE`
Value of **u.ioctl.data**:	New file cache size in units of sectors.

6.42.4.4 Partition FAT cache size

This property type is a FAT cache size in units of sectors. Previous FAT cache is freed before attempting to allocate FAT cache of new size. In case the allocation fails, previous FAT cache is not restored.

u.ioctl.tag:	`CHANFS_PROPERTY_FAT_CACHE_SIZE`
Data type of **u.ioctl.data**:	32-bit unsigned integer
Size of **u.ioctl.data**:	`CHANFS_PROPERTY_SIZE_FAT_CACHE_SIZE`
Value of **u.ioctl.data**:	New partition FAT cache size in units of sectors.

6.42.4.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.42.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_TAG_SET** (5.24.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_setProperty** call.
EFSNULLPOINTER	Parameter 'obj' or 'data' is NULL.
ENOMEM	No memory to allocate a request.

6.42.6 Source Code

The source code of the **sfs_setProperty** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_property.c

6.42.7 Example

```
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  sdd_obj_t * file;
  char label[CHANFS_PROPERTY_SIZE_PARTITION_LABEL];
  uint32_t attributes;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  strcpy(label, "datapart");
  if (sfs_setProperty(partition, CHANFS_PROPERTY_PARTITION_LABEL, label,
                                                            sizeof(label)) < 0) {
    PRINTF ("Could not get partition label \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* Set read-only attribute and clear hidden attribute */
  attributes = CHANFS_ATTRIBUTE_READONLY_SET | CHANFS_ATTRIBUTE_HIDDEN_CLR;

  if (sfs_setProperty(file, CHANFS_PROPERTY_FILE_DIR_ATTRIBUTES, &attributes,
                                                       sizeof(attributes)) < 0) {
    PRINTF ("Could not get file attributes \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }
    .
    .
```

6.43 sfs_size64Bad

6.43.1 Description

The **sfs_size64Bad** function is used to get the total size of the bad blocks in a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE64_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE64_GET_REPLY** message. The calling process will be blocked until this message is received.

6.43.2 Syntax

```
int sfs_size64Bad (
   sdd_obj_t * obj,
   uint64_t *bad
);
```

6.43.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
bad	Returned calculated size of a bad space.

6.43.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.43.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE64_GET** (5.22.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_size64Bad** call.
EFSNULLPOINTER	Parameter 'obj' or 'bad' is NULL.

6.43.6 Source Code

The source code of the **sfs_size64Bad** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.43.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * partition;
uint64_t total, free, used, bad;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_size64Bad(partition, &bad) < 0) {
  PRINTF ("Error getting not available partition size \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.44 sfs_size64Free

6.44.1 Description

The **sfs_size64Free** function is used to get free size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE64_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE64_GET_REPLY** message. The calling process will be blocked until this message is received.

6.44.2 Syntax

```
int sfs_size64Free (
   sdd_obj_t * obj,
   uint64_t *free
);
```

6.44.3 Parameter

obj	Object descriptor. Specifies an object which size shall be returned.
free	Returned calculated size of a free space.

6.44.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.44.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE64_GET** (5.22.2.5).

error code	Return value of **sc_miscErrnoGet** After an **sfs_size64Free** call.
EFSNULLPOINTER	Parameter 'obj' or 'free' is NULL.

6.44.6 Source Code

The source code of the **sfs_size64Free** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.44.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * partition;
uint64_t total, free, used, bad;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_size64Free(partition, &free) < 0) {
  PRINTF ("Error getting free partition size \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.45 sfs_size64Total

6.45.1 Description

The **sfs_size64Total** function is used to get total size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE64_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE64_GET_REPLY** message. The calling process will be blocked until this message is received.

6.45.2 Syntax

```
int sfs_size64Total (
   sdd_obj_t * obj,
   uint64_t *total
);
```

6.45.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
total	Returned calculated size of a total space.

6.45.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.45.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE64_GET** (5.22.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_size64Total** call.
EFSNULLPOINTER	Parameter 'obj' or 'total' is NULL.

6.45.6 Source Code

The source code of the **sfs_size64Total** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.45.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * partition;
uint64_t total, free, used, bad;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_size64Total(partition, &total) < 0) {
  PRINTF ("Error getting total partition size \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.46 sfs_size64Used

6.46.1 Description

The **sfs_size64Used** function is used to get used size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE64_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE64_GET_REPLY** message. The calling process will be blocked until this message is received.

6.46.2 Syntax

```
int sfs_size64Used (
   sdd_obj_t * obj,
   uint64_t *used
);
```

6.46.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
used	Returned calculated size of an used space.

6.46.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.46.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE64_GET** (5.22.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_size64Used** call.
EFSNULLPOINTER	Parameter 'obj' or 'used' is NULL.

6.46.6 Source Code

The source code of the **sfs_size64Used** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.46.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  uint64_t total, free, used, bad;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_size64Used(partition, &used) < 0) {
    PRINTF ("Error getting used partition size \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.47 sfs_sizeBad

6.47.1 Description

The **sfs_sizeBad** function is used to get the total size of the bad blocks in a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE_GET_REPLY** message. The calling process will be blocked until this message is received.

6.47.2 Syntax

```
int sfs_sizeBad (
   sdd_obj_t * obj,
   size_t *bad
);
```

6.47.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
bad	Returned calculated size of a bad space.

6.47.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.47.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE_GET** (5.21.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_sizeBad** call.
EFSNULLPOINTER	Parameter 'obj' or 'bad' is NULL.

6.47.6 Source Code

The source code of the **sfs_sizeBad** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.47.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  size_t total, free, used, bad;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_sizeBad(partition, &bad) < 0) {
    PRINTF ("Error getting not available partition size \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.48 sfs_sizeFree

6.48.1 Description

The **sfs_sizeFree** function is used to get free size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE_GET_REPLY** message. The calling process will be blocked until this message is received.

6.48.2 Syntax

```
int sfs_sizeFree (
   sdd_obj_t * obj,
   size_t *free
);
```

6.48.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
free	Returned calculated size of a free space.

6.48.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.48.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE_GET** (5.21.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_sizeFree** call.
EFSNULLPOINTER	Parameter 'obj' or 'free' is NULL.

6.48.6 Source Code

The source code of the **sfs_sizeFree** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.48.7 Example

```
    .
    .
    .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * partition;
size_t total, free, used, bad;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

if (sfs_sizeFree(partition, &free) < 0) {
  PRINTF ("Error getting free partition size \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

    .
    .
    .
```

6.49 sfs_sizeTotal

6.49.1 Description

The **sfs_sizeTotal** function is used to get total size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE_GET_REPLY** message. The calling process will be blocked until this message is received.

6.49.2 Syntax

```
int sfs_sizeTotal (
   sdd_obj_t * obj,
   size_t *total
);
```

6.49.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
total	Returned calculated size of a total space.

6.49.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.49.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE_GET** (5.21.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_sizeTotal** call.
EFSNULLPOINTER	Parameter 'obj' or 'total' is NULL.

6.49.6 Source Code

The source code of the **sfs_sizeTotal** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.49.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  size_t total, free, used, bad;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_sizeTotal(partition, &total) < 0) {
    PRINTF ("Error getting total partition size \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.50 sfs_sizeUsed

6.50.1 Description

The **sfs_sizeUsed** function is used to get used size of a filesystem object: physical drive, partition, directory or file.

The function sends an **SDD_OBJ_SIZE_GET** message to the controller process of a file object and waits for an **SDD_OBJ_SIZE_GET_REPLY** message. The calling process will be blocked until this message is received.

6.50.2 Syntax

```
int sfs_sizeUsed (
    sdd_obj_t * obj,
    size_t *used
);
```

6.50.3 Parameter

obj	Object descriptor.
	Specifies an object which size shall be returned.
used	Returned calculated size of an used space.

6.50.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.50.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_SIZE_GET** (5.21.2.5).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_sizeUsed** call.
EFSNULLPOINTER	Parameter 'obj' or 'used' is NULL.

6.50.6 Source Code

The source code of the **sfs_sizeUsed** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_size.c

6.50.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * partition;
  size_t total, free, used, bad;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition == NULL) {
    PRINTF ("Could not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_sizeUsed(partition, &used) < 0) {
    PRINTF ("Error getting used partition size \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.51 sfs_timeGet

6.51.1 Description

The **sfs_timeGet** function is used to get the last modification time of a filesystem object.

The function sends an **SDD_OBJ_TIME_GET** message to the controller process of a directory or a file object and waits for an **SDD_OBJ_TIME_GET_REPLY** message. The calling process will be blocked until this message is received.

6.51.2 Syntax

```
int sfs_timeGet (
   sdd_obj_t * obj,
   uint32_t *time
);
```

6.51.3 Parameter

obj	Descriptor of an object.
	Specifies a directory or a file to get a modification time of.
time	Modification time.
	Returned modification time of an object.

6.51.4 Time decoding

For FAT filesystem the returned time is encoded using following format:

bits <31:25>	Year offset from 1980 (0..127)
bits <24:21>	Month (1..12)
bits <20:16>	Day (1..31)
bits <15:11>	Hour (0..23)
bits <10:5>	Minute (0..59)
bits <4:0>	Second / 2 (0..29)

A set of macros is provided for decoding each field of returned **data**:

`CHANFS_OBJECTTIME_EXTRACT_YEAR(data)` – Extract year.

`CHANFS_OBJECTTIME_EXTRACT_MONTH(data)` – Extract month.

`CHANFS_OBJECTTIME_EXTRACT_DAY(data)` – Extract day.

`CHANFS_OBJECTTIME_EXTRACT_HOUR(data)` – Extract hour.

`CHANFS_OBJECTTIME_EXTRACT_MINUTE(data)` – Extract minute.

`CHANFS_OBJECTTIME_EXTRACT_SECOND(data)` – Extract second.

6.51.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.51.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_TIME_GET** (5.25.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_timeGet** call.
EFSNULLPOINTER	Parameter 'obj' or 'time' is NULL.

6.51.7 Source Code

The source code of the **sfs_timeGet** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_time.c

6.51.8 Example

```
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;
  uint32_t time;
  int year, month, day, hour, minute, second;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_timeGet(file, &time) < 0) {
    PRINTF ("Could not get time \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  year   = CHANFS_OBJECTTIME_EXTRACT_YEAR(time);
  month  = CHANFS_OBJECTTIME_EXTRACT_MONTH(time);
  day    = CHANFS_OBJECTTIME_EXTRACT_DAY(time);
  hour   = CHANFS_OBJECTTIME_EXTRACT_HOUR(time);
  minute = CHANFS_OBJECTTIME_EXTRACT_MINUTE(time);
  second = CHANFS_OBJECTTIME_EXTRACT_SECOND(time);

    .
    .
```

6.52 sfs_timeSet

6.52.1 Description

The **sfs_timeSet** function is used to set the last modification time of a filesystem object.

The function sends an **SDD_OBJ_TIME_SET** message to the controller process of a directory or a file object and waits for an **SDD_OBJ_TIME_SET_REPLY** message. The calling process will be blocked until this message is received.

6.52.2 Syntax

```
int sfs_timeSet (
   sdd_obj_t * obj,
   uint32_t time
);
```

6.52.3 Parameter

obj	Descriptor of an object.
	Specifies a directory or a file to set a modification time of.
time	Modification time.
	Modification time to be set on an object.

6.52.4 Time encoding

For FAT filesystem the time has to be encoded in the following format:

bits <31:25>	Year offset from 1980 (0..127)
bits <24:21>	Month (1..12)
bits <20:16>	Day (1..31)
bits <15:11>	Hour (0..23)
bits <10:5>	Minute (0..59)
bits <4:0>	Second / 2 (0..29)

A macro is provided to encode the date/time to FAT filesystem format:

```
CHANFS_OBJECTTIME_CONVERT(year, month, day, hour, minute, second)
```

6.52.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.52.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_OBJ_TIME_SET** (5.26.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_timeSet** call.
EFSNULLPOINTER	Parameter 'obj' or 'time' is NULL.

6.52.7 Source Code

The source code of the **sfs_timeSet** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_time.c

6.52.8 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * file;
  uint32_t time;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  file = sfs_get(chanFs, "/ramdisk0/p2/testdir/testfile.txt", SFS_ATTR_FILE);
  if (file == NULL) {
    PRINTF ("Could not get file object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* set modification time to 2014-06-27 13:14:36 */
  time = CHANFS_OBJECTTIME_CONVERT(2014, 6, 27, 13, 14, 36);

  if (sfs_timeSet(file, time) < 0) {
    PRINTF ("Could not set time \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.53 sfs_umount

6.53.1 Description

The **sfs_umount** message is used to unmount a physical drive or a partition.

The function sends an **SFS_UMOUNT** message to the controller process of a physical drive object and waits for an **SFS_UMOUNT_REPLY** message. The calling process will be blocked until this message is received.

6.53.2 Syntax

```
int sfs_umount (
   sdd_obj_t * drive,
   void *custom,
   size_t custom_size
);
```

6.53.3 Parameter

drive	Drive descriptor.
	Specifies a drive descriptor. A partition or paritions on this drive shall be unmounted.
custom	Unmounting parameters (filesystem dependent).
	Specifies which partition shall be unmounted.
custom_size	Size of unmounting parameters (filesystem dependent).

6.53.4 Unmounting parameters

For FAT filesystem the unmounting parameters are defined as a structure:

```
typedef struct chanfs_umount_s {
  int partition;
} chanfs_umount_t;
```

where:

partition	Partition number to unmount.
1-4	Unmount specified partition. In case a physical drive with single filesystem is mounted (no partitions), value of 1 must be used.
-1	Unmount all mounted filesystems mounted on the physical drive.

6.53.5 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.53.6 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_UMOUNT** (5.41.2.6).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_umount** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate a message.

6.53.7 Source Code

The source code of the **sfs_umount** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_umount.c

6.53.8 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * drive;
  chanfs_umount_t umount;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
  if (drive == NULL) {
    PRINTF ("Could not get physical drive object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* 2nd partition is already mounted. Unmount it. */

  umount.partition = 2;
  if (sfs_umount(drive, &umount, sizeof(umount)) < 0) {
    PRINTF ("Error umounting partition \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.54 sfs_unassign

6.54.1 Description

The **sfs_unassign** function is used to remove a block device from the filesystem.

The function sends an **SFS_UNASSIGN** message to the controller process of a filesystem root object and waits for an **SFS_UNASSIGN_REPLY** message. The calling process will be blocked until this message is received.

6.54.2 Syntax

```
int sfs_unassign (
   const sdd_obj_t * root,
   const char name[],
   int forced,
   int free_objects
);
```

6.54.3 Parameter

root	Root filesystem object descriptor.
	Specifies a root filesystem object descriptor.
name	Name of a block device.
	Specifies a name of a block device which shall be removed from the filesystem.
forced	Forced unassign.
	If non-zero, a forced block device removal is requested. Refer to 3.6 for more information about removing block device from the filesystem.
free_objects	Forced objects freeing.
	If non-zero, a forced objects freeing is requested. When this option is used, application does not have to use **sfs_free** (6.21), to free objects (files, directories, partitions, drive) associated with the device being unassigned. Refer to 3.6 for more information about removing block device from the filesystem. This option can be specified as non-zero only if **forced** parameter is also non-zero.

6.54.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.54.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SFS_UNASSIGN** (5.42.3.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_unassign** call.
EFSNULLPOINTER	Parameter is NULL.
ENOMEM	No memory to allocate message.

6.54.6 Source Code

The source code of the **sfs_unassign** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_unassign.c

6.54.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * drive;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  if (sfs_unassign(chanFs, "ramdisk0", 0, 0) < 0) {
    PRINTF ("Error assigning device to filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
  if (drive != NULL) {
    PRINTF ("Should not get physical drive object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

6.55 sfs_waitAdd

6.55.1 Description

The **sfs_waitAdd** function is used to wait until a physical drive is assigned to the filesystem or a partition (entire physical drive) is mounted.

The function sends an **SDD_MAN_NOTIFY_ADD** message to the controller process of a filesystem root or a physical drive object and waits for an **SDD_MAN_NOTIFY_ADD_REPLY** message. The calling process will be blocked until this message is received.

6.55.2 Syntax

```
int sfs_waitAdd (
   sdd_obj_t * dir,
   const char * name,
   sc_ticks_t tmo
);
```

6.55.3 Parameter

dir	Descriptor of a filesystem root or physical drive.
	Specifies a filesystem root or a physical drive that shall notify about adding a new physical drive or mouting a partition.
name	Name of a new object
	Specifies a name of a physical drive or a partition to wait for.
tmo	Timeout.
	Specifies a timeout when function shall return without success.

6.55.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.55.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_MAN_NOTIFY_ADD** (5.16.3.3, 5.16.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_waitAdd** call.
EFSNULLPOINTER	Parameter is NULL.

6.55.6 Source Code

The source code of the **sfs_waitAdd** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_wait.c

6.55.7 Example

```
  .
  .
  .

sdd_obj_t * man;
sdd_obj_t * chanFs;
sdd_obj_t * drive;
sdd_obj_t * partition;

man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

chanFs = sdd_manGetByName(man, "sdd_chanfs");
if (chanFs == NULL) {
  PRINTF ("Could not find filesystem \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* wait for ramdisk0 drive to be added to the filesystem */
if (sfs_waitAdd(chanFs, "ramdisk0", SC_ENDLESS_TMO) < 0) {
  PRINTF ("Error waiting for ramdisk0 to be added \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
if (drive == NULL) {
  PRINTF ("Could not get physical drive object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

/* wait for 2nd partition to be mounted */
if (sfs_waitAdd(drive, "p2", SC_ENDLESS_TMO) < 0) {
  PRINTF ("Error waiting for partition to be mounted \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
if (partition == NULL) {
  PRINTF ("Could not get partition object \n");
  sc_procKill (SC_CURRENT_PID, 0);
}

  .
  .
  .
```

6.56 sfs_waitRm

6.56.1 Description

The **sfs_waitRm** function is used to wait for a physical drive to be removed from the filesystem or a partition (entire physical drive) to be unmounted.

The function sends an **SDD_MAN_NOTIFY_RM** message to the controller process of a filesystem root or a physical drive object and waits for an **SDD_MAN_NOTIFY_RM_REPLY** message. The calling process will be blocked until this message is received.

6.56.2 Syntax

```
int sfs_waitRm (
   sdd_obj_t * dir,
   const char * name,
   sc_ticks_t tmo
);
```

6.56.3 Parameter

dir	Descriptor of a filesystem root or physical drive.
	Specifies a filesystem root or a physical drive that shall notify about removing a physical drive or unmouting a partition.
name	Name of a new object
	Specifies a name of a physical drive or a partition to wait for.
tmo	Timeout.
	Specifies a timeout when function shall return without success.

6.56.4 Return Value

If the function succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.56.5 Errors

For errors other than listed below or for their possible different meanings, please refer to errors returned by **SDD_MAN_NOTIFY_RM** (5.17.3.3, 5.17.4.3).

error code	Return value of **sc_miscErrnoGet**
	After an **sfs_waitRm** call.
EFSNULLPOINTER	Parameter is NULL.

6.56.6 Source Code

The source code of the **sfs_waitRm** function can be found here:

<installation_folder>\sciopta\<version>\sfs\utils\sfs_wait.c

6.56.7 Example

```
    .
    .
    .

  sdd_obj_t * man;
  sdd_obj_t * chanFs;
  sdd_obj_t * drive;
  sdd_obj_t * partition;

  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  chanFs = sdd_manGetByName(man, "sdd_chanfs");
  if (chanFs == NULL) {
    PRINTF ("Could not find filesystem \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* wait for ramdisk0 drive to be added to the filesystem */
  if (sfs_waitAdd(chanFs, "ramdisk0", SC_ENDLESS_TMO) < 0) {
    PRINTF ("Error waiting for ramdisk0 to be added \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  drive = sfs_get(chanFs, "/ramdisk0", SFS_ATTR_DRIVE);
  if (drive == NULL) {
    PRINTF ("Could not get physical drive object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* wait for 2nd partition to be mounted */
  if (sfs_waitAdd(drive, "p2", SC_ENDLESS_TMO) < 0) {
    PRINTF ("Error waiting for partition to be mounted \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  /* wait for 2nd partition to be unmounted */
  if (sfs_waitRm(drive, "p2", SC_ENDLESS_TMO) < 0) {
    PRINTF ("Error waiting for partition to be unmounted \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

  partition = sfs_get(chanFs, "/ramdisk0/p2", SFS_ATTR_PARTITION);
  if (partition != NULL) {
    PRINTF ("Should not get partition object \n");
    sc_procKill (SC_CURRENT_PID, 0);
  }

    .
    .
    .
```

7 Manual versions

7.1 Manual version 1.11

- Chapter 6.4.1 - Absolute path passed to sfs_chdir must start with "/sdd_chanfs".

7.2 Manual version 1.10

- Updated error lists for messages and functions.

7.3 Manual version 1.9

- Chapter 3.8.12 - Added description of file error and end-of-file indicators.
- Chapter 3.13.6.4 - Added options **FS_NORTC**, **NORTC_MDAY**, **NORTC_MON**, **NORTC_YEAR**.
- Chapter 5.8.2.4 - Changed “size” parameter description of **sdd_fileResize_t** structure.
- Chapter 5.9.2.4 - Added **SDD_FILE_RESIZE64** message.
- Chapter 5.29 - Changed message name from **SFS_FSYNC** to **SFS_FFLUSH**. Changed message structure name from **sfs_fsync_t** to **sfs_fflush_t**.
- Chapter 5.34.2.3 - Changed **sfs_fputc_t** structure definition.
- Chapter 5.35.2.3 - Changed **sfs_fputs_t** structure definition.
- Chapter 6.3 - Added **sfs_clearerr** function.
- Chapter 6.9 - Changed function name from **sfs_close** to **sfs_fclose**.
- Chapter 6.11 - Changed **sfs_feof** function description, syntax, parameter list, return value description and example.
- Chapter 6.12 - Added **sfs_ferror** function.
- Chapter 6.13 - Changed function name from **sfs_fsync** to **sfs_fflush**.
- Chapter 6.14 - Changed **sfs_fgets** syntax, parameter list, return value description and example.
- Chapter 6.17 - Function name changed from **sfs_openByPath** to **sfs_fopen**. Changed function function description, syntax, parameter list, return value description and example.
- Chapter 6.19 - Changed **sfs_fputc** syntax, parameter list, return value description and example.
- Chapter 6.20 - Changed **sfs_fputs** syntax, parameter list, return value description and example.
- Chapter 6.22 - Changes function name from **sfs_read** to **sfs_fread**. Changed function description, syntax, parameter list, return value description and example.
- Chapter 6.23 - Changed function name from **sfs_resize** to **sfs_fresize**. Changed function description, syntax, parameter list, return value description and example.
- Chapter 6.24 - Added **sfs_fresize64** function.
- Chapter 6.25 - Changed function name from **sfs_seek** to **sfs_fseek**. Changed function syntax, return value description and example.

- Chapter 6.26 - Changed function name from **sfs_seek64** to **sfs_fseek64**. Changed function syntax, return value description and example.
- Chapter 6.27 - Changed **sfs_ftell** syntax, parameter list, return value description and example.
- Chapter 6.28 - Changed **sfs_ftell64** syntax, parameter list, return value description and example.
- Chapter 6.29 - Changed function name from **sfs_write** to **sfs_fwrite**. Changed function description, syntax, parameter list, return value description and example.
- Message **SFS_FEOF** has been removed.

7.4 Manual version 1.8

- Chapter 3.8 - Removed “Getting last operation error” point (sfs_ferror function).
- Chapter 3.13.6.1 - Chapter name changed. CHANFS_DEVMAN_PATH option removed.
- Chapter 3.13.6.4.5 - Chapter name changed. LFN_UNICODE and _STRF_ENCODE options removed.
- Chapter 5 - Removed message SFS_FERROR / SFS_FERROR_REPLY.
- Chapter 5.2.4 - Removed message SFS_FERROR / SFS_FERROR_REPLY.
- Chapter 6 - Removed function sfs_ferror.
- Chapter 6.14.2 - Changed sfs_fgets syntax.
- Chapter 6.14.4 - Changed sfs_fgets return value description.
- Chapter 6.14.7 - sfs_fgets example updated to new syntax (see 6.14.2).
- Changed filesystem options names (removed leading uderscore): MULTI_PARTITION, VOLUMES, FS_LOCK, MAX_SS, MIN_SS, FS_READONLY, FS_NOFSINFO, USE_STRFUNC, USE_MKFS, USE_LFN, MAX_LFN, CODE_PAGE, FAT_CACHE_DEFAULT_SIZE

7.5 Manual version 1.7

- Chapter 3.5 - added description of how device driver cannot be used elsewhere while added to the filesystem.
- Chapter 3.13.6.3.5 - added CHANFS_USE_TRAP_INTERFACE option description.
- Chapter 3.13.6.3.7 - added CHANFS_WAIT_FOR_DRIVE_REMOVAL_MS option description.

7.6 Manual version 1.6

- Chapter 3.12 - added filesystem error hook description.
- Chapter 3.13.6.4.1 - SAFE FAT support option added.
- Chapter 3.13.6.3.4 - Option description updated.
- Chapter 4 - SAFE FAT chapter added.
- Chapter 5.33.2.5 - added chanfs_format_t structure version for SAFE FAT.
- Chapter 5.4 - added SDD_DEV_IOCTL message.

- Chapter 6.18.4 - added chanfs_format_t structure version for SAFE FAT.
- Chapter 6.36 - added sfs_ioctl function.
- Chapter 6.40 - sfs_objType macro description added

7.7 Manual version 1.5

- Updated error lists for messages and functions.

7.8 Manual version 1.4

- Chapter 3.5 - description updated (information about read-only mode).
- Chapter 3.6 - added description for forced drive removal.
- Chapter 3.11 - updated accidental drive removal description.
- Chapter 3.13.3.2 - description for required priorities updated.
- Chapter 5.12.3 - content removed, as message is no longer supported by filesystem root object.
- Chapter 5.18.3 - content removed, as message is no longer supported by filesystem root object.
- Chapter 5.27 - SFS_ASSIGN message added.
- Chapter 5.42 - SFS_UNASSIGN message added.
- Chapter 6.2 - informations about sfs_assign function updated.
- Chapter 6.54 - informations about sfs_unassign function updated.
- Updated error lists for messages and functions.

7.9 Manual version 1.0

8 Index