Interface specification Flash File System

Subject: Interface spec.

Revision: 2.4
Author: Erik Christensen
Last Updated: 19/01/2005 11:57
Last Updated By: Mogens Lund
Printout Date: 14/02/2006 03:05
File: D:\FFS_test\doc\FFS_interface.doc

History

Revision 0.4

- New error codes: FFS_INTERNAL_FATAL_ERROR, FFS_USER_ID_ERROR and FFS_WRITE_ERROR have been added.

Revision 0.5

- Minor changes in the function-prototypes.
- FFS_MEMORY_RES_ERROR added to the error codes.
- FFS_MAX_NOF_FILES_EXCEEDED added to the error codes.
- FFS_MAX_NOF_OPEN_FILES_EXCEEDED added to the error codes.

Revision 0.6

- New error codes has been added.
- Minor changes in the function-prototypes.

Revision 0.7

- Function FFS_get_id () added.
- FFS_REQUESTED_MEM_NOT_AVAILABLE added to the error-codes.
- FFS_get_file_info() changed.
- Error code table updated.

Revision 1.0

- New template used.

Revision 1.1

- New functions FFS_findfirst() and FFS_findnext() has replaced FFS_get_id() and FFS_get_id_b y_type().

Revision 1.2

- The functions FFS_findfirst() and FFS_findnext() description changed.
- Added functions to use the FFS with filenames.
- Added callback parameter in FFS_close, FFS_create, FFS_write, FFS_read, FFS_append, FFS_modify, FFS_delete, FFS_rename.
- Added user_id parameter in FFS_open
- Swapped the *dst and offset parameters in FFS_read to align it with FFS_write and FFS_modify
- Added FFS_FALSE, FFS_TRUE, FFS_INVALID_FILENAME to the error code table

Revision 1.3

- The LFA functions FFS_get_ref_LFA(), FFS_load_to_LFA_FN() and FFS_load_to_LFA() has been added.
- FFS_copy() and FFS_copy_fn() has been added.

Revision 1.4

- New interface functions added:
  ulong FFS_get_reserved_space_by_type(ffs_filetype_type type);
  ulong FFS_get_reserved_space_in_global_pool(void);
  ulong FFS_get_used_space(void);
  ulong FFS_get_used_space_by_type(ffs_filetype_type type);
  ulong FFS_get_used_space_in_global_pool(void);

ulong FFS_calculate_file_overhead(ulong filesize, BOOL use_filename);
ffs_error_code_type FFS_ready(void);
- Error code list updated.

Revision 1.5
- New interface functions added:
  FFS_truncate () and FFS_truncate_fn()
- New chapter "Guidelines for optimal use of FFS".

Revision 1.6
- Updated the module content with the new FFS source code files.
- Added FFS_set_drm_attrib() and FFS_set_drm_attrib_fn()

Revision 1.7
- Directory support added.
- New errorcodes.

Revision 1.8
- Added FFS_ffirst_fn() and FFS_fnext_fn.

Revision 1.9
- Added FFS_get_file_pos, FFS_set_file_pos and FFS_get_file_id_from_handle.

Revision 2.0
- Added FFS_ready_cb and FFS_get_block_info.

Revision 2.1
- Added FFS_restore_all_factory_default.

Revision 2.2
- Minor add on in guideline chapter.

Revision 2.3
- Limitations chapter updated.

Revision 2.4
- FFS_get_total_space function added.

Terminology

Glossary

Abbreviations

Contents

Introduction

1.1 Overview

In a Mobile Station (MS) there is a need to stored data in a non-volatile memory, so the data contents is not lost when the MS is switched off. The types of data to be stored non-volatile can be split up in the following groups:

1. Adjustment parameters used to calibrate the MS during production. These parameters have a fixed size and structure.
2. Parameters that can be modified during run-time in normal mode (e.g. MMI settings). These parameters have a fixed size and structure.
3. Application data where size and structure are unknown at compile-time.
4. Application data that will be created deleted dynamically at run-time.

NV data of type 1 and 2 will be handled by an EEPROM emulation driver, and is beyond the scope of this document.

NV data of type 3 and 4 is to be handled by a Flash File System (FFS), which is described in this document.

Examples of items that could be stored in the FFS:

- SMS
- Phonebook
- Calendar
- E-mail
- Address book
- WAP profiles
- Ringing tones
- Bitmaps
- Animations

1.2 Features

- Handling of multiple files in a mobile station
- Simple function-interface towards the application level.
- Mechanisms to handle multiple files opened for read/write simultaneously.
- Mechanisms to handle multiple processes with simultaneous FFS-access.
- Files can be predefined at compile-time, or be created/deleted dynamically at run-time
- Power Loss Recovery algorithms to prevent corruption after unintended power loss.Directory structures.
- Support of the following memory medias:
  - Single bank flash types (combined code-storage and FFS data in the same device).
  - Dual bank flash types (combined code-storage and FFS data in the same device).
  - Separate parallel data flash types.
  - Can be adapted to serial data flash types, Smart Media cards etc.

1.3 Limitations

- Max. number of files: 0-65536 (configurable).
- Max. number of filetypes: 0-256 (configurable).
- Max. file-size: 4 Giga bytes.
- Max. size FFS storage: 4 Giga bytes.

1.4 Module contents

File	Description
FFS_INTERFACE.C	Contains all the source code to the FFS interface function calls
FFS.C	Contains all the source code for the core functionality of FFS
FFS.H	Specifies the interface to the FFS.
FFS_DEF.H	Contains configuration settings to customize the FFS.
FFS_OPCODES.H	Specifies all the AT# commands understood by FFS
FFS_INTERNAL.H	Specifies all variables shared between the FFS files.
FFS_TYPES.H	Defines the variable types used in the FFS code

1.5 Guidelines for optimal use of FFS

1.5.1 When creating files.

Guidelines for optimal usage of the FFS interface, and minimum generation of garbage in the system when creating new files:

When creating new files the following FFS interface functions should be used:

```
FFS_create() or FFS_create_fn()   //Here filesize is given as input parameter

FFS_write() or FFS_write_fn()     //Could be called N times until the file is written.

FFS_close()                       //Will save the file
```

Above procedure will not generate any garbage in FFS at all, and the access speed to and from FFS will also be improved dramatically by following this guideline.

The procedure that might be used by some applications is like below:

```
FFS_create() or FFS_create_fn()   //Here filesize is set to 0

FFS_close()                       //Will save the file

FFS_open() or FFS_open_fn()       //Open in append mode

FFS_append() or FFS_append_fn()   //Is called in a loop until the right filesize

FFS_close()

FFS_open() or FFS_open_fn()       //Open in modify mode

FFS_modify() or FFS_modify_fn()   //Is now used to write the actual filedata to the file

FFS_close()
```

In the above procedure, both the append- and modify-operations will generate garbage. Especially the modify part since all data in the file will be replaced in new sectors, but also some index sectors to keep track on the location of data sectors will need to be replaced with new sectors.

If all applications could use the first method a lot of garbage generation could be avoided.
This of course requires that the file size is known initially. If only an approximate file size is known this procedure could be used also. It just means the resulting file size should be trimmed at the end for the file update. Either to extend the file size by FFS_append() or make the file smaller with FFS_truncate().

If the optimal solution without garbage generation is not achievable for some reason (e.g. unknown file-size when creating), the following sequence can be used:

```
FFS_create() or FFS_create_fn()   //Here filesize is set to 0

FFS_close()                       //Will save the file

FFS_open() or FFS_open()          //Open in append mode

FFS_append() or FFS_append_fn()   //Is called in a loop writting the
                                  //actual filedata until the EOF
FFS_close()
```

By doing this, the modify operation is avoided. The modify operation is the worst garbage generator.

Chunk size:
In general, it is good practice to do all file accesses in as big chunks as possible. This will increase the performance, since the overhead from context switching and bookkeeping algorithms will be reduced.

1.5.2 When modifying existing files.

When the entire contents or a big portion of a file is being modified, it is good practice to do it in as big chunks as possible. For instance, modify some hundred bytes or a few Kbytes for each FFS modify request. By doing this, the amount of garbage generated would be kept on a minimum level.

When requesting the FFS modify operation and only updating a few bytes a lot of garbage will be generated.
The smallest allocation unit in FFS is a sector (normally 256 bytes).
This means that a new data sector (256 bytes) should be allocated, and at least the file info sector (also 256 bytes) has to be rewritten too. For big files there is a number of index sectors which also has to be rewritten.
A minimum of 512 bytes of garbage is generated for each modify-operation requested. For big files it will be more.

The access speed to and from FFS will also be improved dramatically by following this guideline.

1.5.3 FFS user IDs.

Each RTOS process accessing FFS needs to have a unique FFS user ID defined in the user ID table. One specific FFS user can only have one active operation in FFS, so if N processes uses the same user-ID when accessing FFS, these processes will be blocking each other.

1.5.4 Memory pool design.

The FFS offers the possibility to assign a dedicated memory pool to each file-type defined in FFS. These dedicated pools can only be used for files of this specific type, so the dedicated pool provides a minimum amount of available memory for the given file-type at any time. If more memory than specified in the dedicated pool is needed for a specific file-type during run-time, the extra memory will be allocated from the global pool.
To obtain a flexible memory design these dedicated pools should be as small as possible, and the global pool should be as big as possible. A small global pool could lead to trouble because all garbage is placed in the global pool, and if there is a lot of garbage generating operations there is a risk that the global pool gets full, and it’s necessary to do garbage collection before further file-operations could be carried out.

1.6 Interface specification

1.6.1 Function interface

When using callback functions, will all errors detected by checking the input parameters be returned immediately when the interface function returns. If the input parameters are OK then the interface function will activate the FFS and return FFS_SUCCESS.
The result of the FFS operation will be returned with the callback function.

1.6.1.1 FFS_initialize()

Prototype:	ffs_error_code_type **FFS_initialize**(void)
Input parameters:	None
Output parameters:	Error code for the operation.
Description:	After power up the FFS_initialize() must be called. In this initialization there will be a check for if the last power off was a "structured power off" or there is a need to do some clean up. The FFS operations will not be accessible before the initialization has completed.

1.6.1.2 FFS_deactivate()

Prototype:	void **FFS_deactivate**(void)
Input parameters:	None
Output parameters:	None
Description:	Should be called before a structured power off is carried out.

1.6.1.3 FFS_ready()

Prototype: ffs_error_code_type **FFS_ready**(void)

Input parameters: None

Output parameters: True if FFS has finished initialization.

Description: After power up the FFS initializes itself by performing power loss recovery, locating all files stored in FFS and restores factory files if necessary. This can take some time and FFS_ready can be used to check if FFS has finished this initialization.

1.6.1.4 FFS_ready_cb()

Prototype: ffs_error_code_type FFS_ready_cb(ffs_user_type user_id, ffs_callback_type, void *client_arg)

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.
*client_arg:	Extended callback functionality. Using the callback as argument.

Output parameters: Error code for the operation.

Description: After power up the FFS initializes itself by performing power loss recovery, locating all files stored in FFS and restores factory files if necessary. This can take some time and FFS_ready_cb can be used to check if FFS has finished this initialization.

1.6.1.5 FFS_get_version()

Prototype: ffs_error_code_type **FFS_get_version**(void)

Input parameters: None

Output parameters: Version number of FFS.

Description: Returns the version number of FFS.

1.6.1.6 FFS_get_revision()

Prototype: ffs_error_code_type **FFS_get_revision**(void)

Input parameters: None

Output parameters: Revision number of FFS.

Description: Returns the revision number of FFS.

1.6.1.7 FFS_open()

Prototype:

```
ffs_error_code_type FFS_open(ffs_user_type user_id,
                             word id,
                             ffs_handle_type *hp,
                             ffs_open_file_mode_type mode)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
hp:	A pointer to the location where to store the handle.
mode:	Specifies in which mode to open the file.

Output parameters: Error code for the operation.

Description: Opens a file with the specified id for operations specified by *mode*. Only possible if the file exist and is not already open. Stores the file-handle at the location specified by *hp.

1.6.1.8 FFS_open_fn()

Prototype:

```
ffs_error_code_type FFS_open_fn(ffs_user_type user_id,
                                char *filename,
                                ffs_handle_type *hp,
                                ffs_open_file_mode_type mode)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
hp:	A pointer to the location where to store the handle.
mode:	Specifies in which mode to open the file.

Output parameters: Error code for the operation.

Description: Opens a file with the specified id for operations specified by *mode*. Only possible if the file exist and is not already open. Stores the file-handle at the location specified by *hp.

1.6.1.9 FFS_close()

Prototype:

```
ffs_error_code_type FFS_close(ffs_user_type user_id,
                              word id,
                              ffs_handle_type h,
                              void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
h:	The file handle.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Closes a file with the specified id. Only possible if the file exists and is in the open state. Saves the specified file if the file was opened for write/modify/append operations.

1.6.1.10 FFS_close_fn()

Prototype:

```
ffs_error_code_type FFS_close_fn(ffs_user_type user_id,
                                 char *filename,
                                 ffs_handle_type h,
                                 void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
h:	The file handle.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Closes a file with the specified id. Only possible if the file exists and is in the open state. Saves the specified file if the file was opened for write/modify/append operations.

1.6.1.11 FFS_copy()

Prototype: ffs_error_code_type **FFS_copy**(ffs_user_type user_id,
word src_id,
word dst_id,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
src_id:	The source file id number.
dst_id:	The destination file id number.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies a file with the source id to the destination id. Only possible if the file exists and is closed. Copy will open the source file, create a new file with the destination id, read the source file data and write it to the newly created file.

1.6.1.12 FFS_copy_fn()

Prototype: ffs_error_code_type **FFS_copy_fn**(ffs_user_type user_id,
char *src_name,
char *dst_name,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
src_name:	The source file name.
dst_name:	The destination file name.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies a file with the source file name to the destination file name. Only possible if the file exists and is closed. Copy will open the source file, create a new file with the destination file name, read the source file data and write it to the newly created file.

1.6.1.13 FFS_create()

Prototype:

```
ffs_error_code_type FFS_create(ffs_user_type user_id,
                               word id,
                               ffs_handle_type *hp,
                               ffs_filetype_type type,
                               ulong size,
                               void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number of the file to be created.
hp:	A pointer to the location where to store the handle.
type:	Specifies which memory-pool to allocate from.
size:	The maximum file size.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Allocates space for a new file of the given type, and the file can be opened for write-operations. Stores the file-handle at the location specified by *hp.

1.6.1.14 FFS_create_fn()

Prototype:

```
ffs_error_code_type FFS_create_fn(ffs_user_type user_id,
                                  char *filename,
                                  ffs_handle_type *hp,
                                  ffs_filetype_type type,
                                  ulong size,
                                  void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file to be created.
hp:	A pointer to the location where to store the handle.
type:	Specifies which memory-pool to allocate from.
size:	The maximum file size.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Allocates space for a new file of the given type, and the file can be opened for write-operations. Stores the file-handle at the location specified by *hp.

1.6.1.15 FFS_write()

Prototype: ffs_error_code_type **FFS_write**(ffs_user_type user_id,
word id,
ffs_handle_type h,
byte *src,
ulong offset,
ulong nof,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
offset:	The offset in the file where to write.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies the data from the specified location in RAM to the specified location in the file. Only possible if the file exists and is opened for write-operations.

1.6.1.16 FFS_write_fn()

Prototype: ffs_error_code_type **FFS_write_fn**(ffs_user_type user_id,
char *filename,
ffs_handle_type h,
byte *src,
ulong offset,
ulong nof,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
offset:	The offset in the file where to write.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies the data from the specified location in RAM to the specified location in the file. Only possible if the file exists and is opened for write-operations.

1.6.1.17 FFS_read()

Prototype:

```
ffs_error_code_type FFS_read(ffs_user_type user_id,
                             word id,
                             ffs_handle_type h,
                             byte *dst,
                             ulong offset,
                             ulong nof,
                             void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
h:	The file handle.
dst:	Pointer to the local store where to copy data.
offset:	The offset in the file where to read.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies the data from the specified file to the dst location in RAM. Only possible if the file exists and is opened for read-operations.

1.6.1.18 FFS_read_fn()

Prototype:

```
ffs_error_code_type FFS_read_fn(ffs_user_type user_id,
                                char *filename,
                                ffs_handle_type h,
                                byte *dst,
                                ulong offset,
                                ulong nof,
                                void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
h:	The file handle.
dst:	Pointer to the local store where to copy data.
offset:	The offset in the file where to read.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies the data from the specified file to the dst location in RAM. Only possible if the file exists and is opened for read-operations.

1.6.1.19 FFS_append()

Prototype:

```
ffs_error_code_type FFS_append (ffs_user_type user_id,
                                word id,
                                ffs_handle_type h,
                                byte *src,
                                ulong nof,
                                void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number of the file to be appended.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Allocates extra space for an existing file, and adds the specified data in the end of the existing file.

1.6.1.20 FFS_append_fn()

Prototype:

```
ffs_error_code_type FFS_append_fn(ffs_user_type user_id,
                                  char *filename,
                                  ffs_handle_type h,
                                  byte *src,
                                  ulong nof,
                                  void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Allocates extra space for an existing file, and adds the specified data in the end of the existing file.

1.6.1.21 FFS_truncate()

Prototype: ffs_error_code_type **FFS_truncate** (ffs_user_type user_id,
word id,
ulong new_size,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number of the file to be appended.
new_size:	The new size of the file.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Truncate the file to the new size by discarding data from the end of the file.

1.6.1.22 FFS_truncate_fn()

Prototype: ffs_error_code_type **FFS_truncate_fn**(ffs_user_type user_id,
char *filename,
ulong new_size,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
new_size:	The new size of the file.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Truncate the file to the new size by discarding data from the end of the file.

1.6.1.23 FFS_modify()

Prototype: ffs_error_code_type **FFS_modify**(ffs_user_type user_id,

```
ffs_error_code_type FFS_modify(ffs_user_type user_id,
                               word id,
                               ffs_handle_type h,
                               byte *src,
                               ulong offset,
                               ulong nof,
                               void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
offset:	The offset in the file where to write.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: An existing file can be modified by this operation. The whole file or a part of the file can be modified.

1.6.1.24 FFS_modify_fn()

Prototype:

```
ffs_error_code_type FFS_modify_fn(ffs_user_type user_id,
                                  char *filename,
                                  ffs_handle_type h,
                                  byte *src,
                                  ulong offset,
                                  ulong nof,
                                  void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
h:	The file handle.
src:	Pointer to the local store where to copy data from.
offset:	The offset in the file where to write.
nof:	Number of bytes to copy.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: An existing file can be modified by this operation. The whole file or a part of the file can be modified.

1.6.1.25 FFS_set_drm_attrib()

Prototype: ffs_error_code_type **FFS_set_drm_attrib**(ffs_user_type user_id,
word id,
ffs_drm_attrib_type *drm_attrib,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
drm_attrib:	Pointer to the local store where to copy the drm attributes from.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: The default values for the DRM attributes can be changed by this operation for an existing file. The file must be opened for setting DRM attributes.

1.6.1.26 FFS_set_drm_attrib_fn()

Prototype: ffs_error_code_type **FFS_set_drm_attrib_fn**(ffs_user_type user_id,
char *filename,
ffs_drm_attrib_type *drm_attrib,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
drm_attrib:	Pointer to the local store where to copy the drm attributes from.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: The default values for the DRM attributes can be changed by this operation for an existing file. The file must be opened for setting DRM attributes.

1.6.1.27 FFS_delete()

Prototype: ffs_error_code_type **FFS_delete**(ffs_user_type user_id,
word id,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number of the file to be created.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Marks the specified file as "deleted" in the FFS, and any further operations on the file will be rejected. The file will be cleaned-up by the garbage collector (background task). The file should not be open when the delete operation is requested.

1.6.1.28 FFS_delete_fn()

Prototype: ffs_error_code_type **FFS_delete_fn**(ffs_user_type user_id,
char *filename,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Marks the specified file as "deleted" in the FFS, and any further operations on the file will be rejected. The file will be cleaned-up by the garbage collector (background task). The file should not be open when the delete operation is requested.

1.6.1.29 FFS_rename()

Prototype: ffs_error_code_type **FFS_rename**(ffs_user_type user_id,
word id,
word new_id,
ffs_filetype_type new_type,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number of the file to be renamed.
new_id:	The file id number to be renamed to.

new_type:	Specifies the file-type to be renamed to.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: An existing file with a specified *id* can be renamed to *new_id* and be converted to another file-type. Is useful when using temporary files. Not possible if the file is in the open state.

1.6.1.30 FFS_rename_fn()

Prototype:

```
ffs_error_code_type FFS_rename_fn(ffs_user_type user_id,
                                  word id,
                                  word new_id,
                                  ffs_filetype_type new_type,
                                  void (*callback)( ffs_error_code_type result))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
new_id:	The file id number to be renamed to.
new_type:	Specifies the file-type to be renamed to.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: An existing file with a specified *id* can be renamed to *new_id* and be converted to another file-type. Is useful when using temporary files. Not possible if the file is in the open state.

1.6.1.31 FFS_get_next_free_id()

Prototype: word **FFS_get_next_free_id** (void)

Input parameters: None.

Output parameters: Nnext free *id*, 0xffff = error.

Description: Returns the next free *id* in the FAT or 0xffff if no *id* is available. Only used when there is no predefined *id* for the file to be created.

1.6.1.32 FFS_file_exist().

Prototype: boolean **FFS_file_exist** (word id)

Input parameters: id: The file id number.

Output parameters: True if file already exists, otherwise false.

Description: Checks the FFS for the presence of the file with the specified *id.*

1.6.1.33 FFS_file_exist_fn().

Prototype: boolean **FFS_file_exist_fn** (ffs_user_type user_id,
char *filename)

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.

Output parameters: True if file already exists, otherwise false.

Description: Checks the FFS for the presence of the file with the specified *id.*

1.6.1.34 FFS_findfirst()

Prototype: boolean **FFS_findfirst** (word *id_ptr,
ffs_filetype_type type)

Input parameters:

id_ptr:	Specifies where to store the found Id.
type:	Filetype to search for.

Output parameters: True if a file is found, otherwise false.

Description: Returns the first stored file of the specified type.
Use FFS_ALL_FILETYPES to find files of all types.

1.6.1.35 FFS_findfirst_fn()

Prototype: boolean **FFS_findfirst_fn**(ffs_user_type user_id,
char *filename_buffer,
char *filespec,,
ffs_filetype_type type)

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename_buffer:	Pointer to the local store where to copy the found name.
filespec:	Specification of file to find. Accepts the '*' and '?' wild cards
type:	Filetype to search for.

Output parameters: True if a file is found, otherwise false.

Description: Returns the first stored file of the specified type.
Use FFS_ALL_FILETYPES to find files of all types.

1.6.1.36 FFS_findnext()

Prototype: boolean **FFS_findnext**(word last_id,
word *id_ptr,

ffs_filetype_type type)

Input parameters: last_id: The last Id found by findfirst or findnext.
id_ptr: Specifies where to store the found Id. type: Filetype to search for.

Output parameters: True if a file is found, otherwise false.

Description: Returns the next stored file of the specified type.
Use FFS_ALL_FILETYPES to find files of all types.

1.6.1.37 FFS_findnext_fn()

Prototype:

```
boolean FFS_findnext_fn(ffs_user_type user_id,
                        char *filename_buffer,
                        char *filespec,,
                        ffs_filetype_type type)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename_buffer:	Pointer to the local store where to copy the found name.
filespec:	Specification of file to find. Accepts the '*' and '?' wild cards
type:	Filetype to search for.

Output parameters: True if a file is found, otherwise false.

Description: Returns the next stored file of the specified type.
Use FFS_ALL_FILETYPES to find files of all types.

1.6.1.38 FFS_get_file_info().

Prototype:

```
ffs_error_code_type FFS_get_file_info (ffs_user_type user_id,
                                       word id,
                                       ffs_file_info_type *info)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
*info:	Pointer to the file info record where to store data.

Output parameters: Error code for the operation.

Description: Returns a file info record containing the following info about the file: *type*, *size* and *timestamp*.

1.6.1.39 FFS_get_file_info_fn().

Prototype:

```
ffs_error_code_type FFS_get_file_info_fn (ffs_user_type user_id,
                                          char *filename,
```

ffs_file_info_type *info)

Input parameters: user_id: Identifies the user process and the corresponding FFS priority.
filename: Name of the file.
*info: Pointer to the file info record where to store data.

Output parameters: Error code for the operation.

Description: Returns a file info record containing the following info about the file: *type*, *size* and *timestamp*.

1.6.1.40 FFS_get_file_name().

Prototype: ffs_error_code_type **FFS_get_file_name** (ffs_user_type user_id,
word id,
ffs_file_info_type *filename_buffer)

Input parameters: user_id: Identifies the user process and the corresponding FFS priority.
id: The file id number.
*info: Pointer to a buffer where to store the filename.

Output parameters: Error code for the operation.

Description: If the identified by the id have a filename, then the function returns the filename. The buffer must be big enough to store the filename.

1.6.1.41 FFS_get_file_id().

Prototype: ffs_error_code_type **FFS_get_file_id** (ffs_user_type user_id,
char *filename,
word *id)

Input parameters: user_id: Identifies the user process and the corresponding FFS priority.
filename: Name of the file.
*id: Pointer where to store the id.

Output parameters: Error code for the operation.

Description: Returns the id of the file identified by the filename.

1.6.1.42 FFS_get_file_id_from_handle().

Prototype: ffs_error_code_type **FFS_get_file_id_from_handle**(ffs_user_type user_id,
ffs_handle_type handle,
word *id)

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	handle:	The file handle.
	*id:	Pointer where to store the id.

Output parameters: Error code for the operation.

Description: Returns the file id from the file handle. If error, file id 0xFFFF is returned

1.6.1.43 FFS_set_file_pos ().

Prototype: ffs_error_code_type **FFS_set_file_pos** (ffs_user_type user_id,
ffs_handle_type handle,
unsigned long file_pos,
void (*callback)(ffs_error_code_type result))

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	handle:	The file handle.
	file_pos:	The new file pos.
	callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Changes the file's position in streaming read mode.

1.6.1.44 FFS_get_file_pos ().

Prototype: ffs_error_code_type **FFS_get_file_pos** (ffs_user_type user_id,
ffs_handle_type handle,
unsigned long file_pos,
void (*callback)(ffs_error_code_type result))

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	handle:	The file handle.
	file_pos:	Pointer where to store the file pos.
	callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Captures the file's read position in streaming mode.

1.6.1.45 FFS_get_nof_files ()

Prototype: word **FFS_get_nof_files** (void)

Input parameters: None.

Output parameters: The number of files stored.

Description: Returns the number of stored files in the FFS.

1.6.1.46 FFS_get_nof_files_by_type ()

Prototype: word **FFS_get_nof_files_by_type** (ffs_filetype_type type)

Input parameters: type: Filter files of specified type only.

Output parameters: The number of files stored.

Description: Returns the number of stored files of the specified type.

1.6.1.47 FFS_get_reserved_space_by_type ()

Prototype: ulong **FFS_get_reserved_space_by_type** (ffs_filetype_type type)

Input parameters: type: Filter files of specified type only.

Output parameters: The number of bytes originally reserved to this filetype.

Description: Returns the number of bytes originally reserved to this filetype.

1.6.1.48 FFS_get_reserved_space_in_global_pool ()

Prototype: ulong **FFS_get_reserved_space_in_global_pool** (void)

Input parameters: None.

Output parameters: The number of bytes originally reserved to the global pool.

Description: Returns the number of bytes originally reserved to the global pool. This is all the space not reserved to any specific filetype. This is the same as **FFS_get_reserved_space_by_type** (FFS_GENERIC_FILETYPE).

1.6.1.49 FFS_get_total_space ()

Prototype: ulong **FFS_get_total_space** (void)

Input parameters: None.

Output parameters: The total FFS space in bytes.

Description: Returns the total number of bytes in the FFS storage area.

1.6.1.50 FFS_get_used_space ()

Prototype: ulong **FFS_get_used_space** (void)

Input parameters: None.

Output parameters: The number of used bytes.

Description: Returns the total number of used bytes in FFS. Space filled with garbage is not considered used because it will be freed when the garbage collector cleans up.

1.6.1.51 FFS_get_used_space_by_type ()

Prototype: ulong **FFS_get_used_space_by_type** (ffs_filetype_type type)

Input parameters: type: Filter files of specified type only.

Output parameters: The number of used bytes in the specific memory pool.

Description: Returns number of used bytes in the specific memory pool.

1.6.1.52 FFS_get_used_space_in_global_pool ()

Prototype: ulong **FFS_get_used_space_in_global_pool** (void)

Input parameters: None.

Output parameters: The number of used bytes in the global memory pool.

Description: Returns the number of used bytes in the global memory pool. This includes the space used by specific file types exceeding their own reservations.

1.6.1.53 FFS_get_free_space ()

Prototype: ulong **FFS_get_free_space** (void)

Input parameters: None.

Output parameters: The number of free bytes.

Description: Returns the total number of available bytes free in the FFS. Space filled with garbage is not available before the garbage collector has cleaned up.

1.6.1.54 FFS_get_free_space_by_type ()

Prototype: ulong **FFS_get_free_space_by_type** (ffs_filetype_type type)

Input parameters: type: Filter files of specified type only.

Output parameters: The maximum size of one file using all of the available bytes free in the memory pool.

Description: Returns the maximum size of one file using all of the available bytes free in the memory pool specified by *type*. This doesn't include the free space in the global pool although this space also can be used to create files of a specific type. To include this add the value returned by FFS_get_free_space_in_global_pool().

1.6.1.55 FFS_get_free_space_in_global_pool ()

Prototype: ulong **FFS_get_free_space_in_global_pool** (void)

Input parameters: None.

Output parameters: The maximum size of one file using all of the available bytes free in the global memory pool.

Description: Returns the maximum size of one file using all of the available bytes free in the global memory pool.
This is the same as **FFS_get_free_space_by_type** (FFS_GENERIC_FILETYPE).

1.6.1.56 FFS_get_raw_free_space_by_type ()

Prototype: ulong **FFS_get_raw_free_space_by_type** (ffs_filetype_type type)

Input parameters: type: Filter files of specified type only.

Output parameters: The number of bytes free in the memory pool.

Description: Returns the number of bytes free in the memory pool specified by *type*.

1.6.1.57 FFS_get_raw_free_space_in_global_pool ()

Prototype: ulong **FFS_get_raw_free_space_in_global_pool** (void)

Input parameters: None.

Output parameters: The number of bytes free in the global memory pool.

Description: Returns the number of bytes free in the global memory pool. This is the same as **FFS_get_raw_free_space_by_type** (FFS_GENERIC_FILETYPE).

1.6.1.58 FFS_calculate_file_overhead ()

Prototype: ulong **FFS_calculate_file_overhead** (ulong filesize, BOOL use_filename)

Input parameters:

filesize:	The size of the file.
use_filename:	Indicates if the file has a filename. 0 = no filename, >0 = has filename

Output parameters: The number of bytes added because of overhead.

Description: The actual space occupied by a file is the filesize + the overhead of the file. This function returns the overhead of a file with a given filesize. The optional filename is also overhead.

1.6.1.59 FFS_get_LFA_ref ()

Prototype:

```
ffs_error_code_type FFS_get_LFA_ref (ffs_user_type user_id,
                                      LFA_id_type lfa_id,
                                      void **ref,
                                      ulong *size)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
lfa_id:	The LFA id type that identifies the file.
ref:	The reference pointer that will be returned to point to the file data.
size:	Data size.

Output parameters: Error code for the operation.

Description: Returns a pointer to data and a data size from a file copied from the FFS to the LFA memory area where the file data is placed aligned in memory. The reference pointer gets NULL if the file with the given lfa_id doesn’t exist.

1.6.1.60 FFS_load_to_LFA ()

Prototype:

```
ffs_error_code_type FFS_load_to_LFA (ffs_user_type user_id,
                                      word id,
                                      LFA_id_type lfa_id,
                                      void **ref,
                                      ulong *size,
                                   void (*callback)(ffs_error_code_type result,
                                                      result,
                                                         void **ref, ulong *size))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
id:	The file id number.
lfa_id:	The LFA id type that identifies the file.
ref:	The reference pointer that will be returned to point to the file data.
size:	Data size.
callback:	Function to use when returning the result of the operation by call back.

Output parameters: Error code for the operation.

Description: Reads the file from the flash file system and copies it to the linear file access memory area specified with the lfa_id. If the specified linear file access memory is already in use, then the used memory area is erased before the new file is written. If the specified LFA memory area doesn't exist, then a NULL reference pointer is returned. Only possible if the file exist and is closed.

1.6.1.61 FFS_load_to_LFA_FN ()

Prototype:

```
ffs_error_code_type FFS_load_to_LFA_FN (ffs_user_type user_id,
                                        char *filename,
                                        LFA_id_type lfa_id,
                                        void **ref ,
                                        ulong *size,
                             void (*callback)(ffs_error_code_type result,
                                                 void **ref, ulong *size ))
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
filename:	Name of the file.
lfa_id:	The LFA id type that identifies the file.
ref:	The reference pointer that will be returned to point to the file data.
size:	Data size.
callback:	Function to use when returning the result of the operation by call back.

Output parameters: Error code for the operation.

Description: Reads the file from the flash file system and copies it to the linear file access memory area specified with the lfa_id. If the specified linear file access memory is already in use, then the used memory area is erased before the new file is written. If the specified LFA memory area doesn't exist, then a NULL reference pointer is returned. Only possible if the file exist and is closed.

1.6.1.62 FFS_get_LFA_info ()

Prototype:

```
ffs_error_code_type FFS_get_LFA_info (ffs_user_type user_id,
                                      LFA_id_type lfa_id,
```

ffs_file_info_LFA_type *info)

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	lfa_id:	The LFA id type that identifies the file.
	info:	Info of the file in the LFA memory area fetched at the FFS with the file id.

Output parameters: Error code for the operation.

Description: Returns info of the file placed at the LFA memory. The info is fetched at the FFS where the original file is placed. Only possible if the originally copied file still exist at the FFS.

1.6.1.63 FFS_restore_factory_default ()

Prototype: ffs_error_code_type **FFS_restore_factory_default** (ffs_user_type user_id,
word id,
void (*callback)(ffs_error_code_type result))

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	id:	The file id number of the file to be restored. This can be the id of either the static or the dynamic file.
	callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: This function will delete the dynamic copy of the static factory file if the copy exists and is modified compared to the static file. Then it will make a new copy of the factory file.

1.6.1.64 FFS_restore_factory_default_fn ()

Prototype: ffs_error_code_type **FFS_restore_factor_default_fn** (ffs_user_type user_id,
char *filename,
void (*callback)(ffs_error_code_type result))

Input parameters:	user_id:	Identifies the user process and the corresponding FFS priority.
	filename:	The name of the file to be restored. This can be the name of either the static or the dynamic file.
	callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: This function will delete the dynamic copy of the static factory file if the copy exists and is modified compared to the static file. Then it will make a new copy of the factory file.

1.6.1.65 FFS_restore_all_factory_default ()

Prototype: ffs_error_code_type **FFS_restore_all_factory_default** (ffs_user_type user_id,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: This function will restore all the factory files it can find. The procedure of restoring each file is the same as in FFS_restore_factory_default().

1.6.1.66 FFS_mkdir ()

Prototype: ffs_error_code_type **FFS_mkdir**(ffs_user_type user_id,
const char *path,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
path:	The path name of the directory to be created.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Creates a new directory with the name specified by path.

1.6.1.67 FFS_rmdir ()

Prototype: ffs_error_code_type **FFS_rmdir**(ffs_user_type user_id,
const char *path,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
path:	The path name of the directory to be deleted.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Deletes the directory with the name specified by path.

1.6.1.68 FFS_isdir ()

Prototype: ffs_error_code_type **FFS_isdir**(ffs_user_type user_id,
const char *path,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
path:	The path name of the directory/file to be checked.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Checks if the path specifies a directory or a file.

1.6.1.69 FFS_setcwd ()

Prototype: ffs_error_code_type **FFS_**setcwd (ffs_user_type user_id,
const char *path,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
path:	Specifies the location where to read the new CWD path.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Changes the current working directory to path.

1.6.1.70 FFS_getcwd ()

Prototype: ffs_error_code_type **FFS_g**etcwd (ffs_user_type user_id,
char *path,
void (*callback)(ffs_error_code_type result))

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
path:	Specifies the location where to store the CWD path.
callback:	Function to use when returning the result of the operation. If it is NULL then the result is returned at return of the interface function.

Output parameters: Error code for the operation.

Description: Copies the full name of the current working directory to path.

1.6.1.71 FFS_ffirst_fn()

Prototype:

```
boolean FFS_ffirst_fn(ffs_user_type user_id,
                      char *name_buffer,
                      char *name_spec,
                      ffs_ref_type *name_ref,
                      ffs_filetype_type type)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
name_buffer:	Pointer to the local store where to copy the found dir or file name.
name_spec:	Search specification. Accepts the '*' and '?' wild cards
name_ref:	Internal reference to the located dir/file. O parameter.
type:	Filetype to search for.

Output parameters: True if a dir/file is found, otherwise False.

Description: Returns data in the "name_ref" buffer for the first found dir/file.

1.6.1.72 FFS_fnext_fn()

Prototype:

```
boolean FFS_fnext_fn(ffs_user_type user_id,
                     char *name_buffer,
                     char *name_spec,
                     ffs_ref_type *name_ref,
                     ffs_filetype_type type)
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
name_buffer:	Pointer to the local store where to copy the found dir or file name.
name_spec:	Search specification. Accepts the '*' and '?' wild cards
name_ref:	Internal reference to the located dir/file. I/O parameter.
type:	Filetype to search for.

Output parameters: True if a dir/file is found, otherwise False.

Description: When calling FFS_fnext_fn() the user shall give a pointer to the "name_ref" data last returned by FFS. FFS uses "name_ref" to find the correct place to continue the search. The next found dir/file ref is copied to "name_ref" by FFS.

1.6.1.73 FFS_get_block_info().

Prototype:

```
ffs_error_code_type FFS_get_block_info(ffs_user_type user_id,
                                       ffs_block_info_type *block_info) {
```

Input parameters:

user_id:	Identifies the user process and the corresponding FFS priority.
*block_info:	Pointer to the block info record where to store data.

Output parameters: Error code for the operation.

Description: Returns a block info record containing the following info about the block: lbn, bec, bf, gc, addr and number of sectors in block.

1.7 Error codes.

The possible error codes that can be returned from the function interface are listed below.

Error code	Description
FFS_FALSE	0 Used by functions which can return a BOOL result or a error code in a callback function.
FFS_TRUE	1 Used by functions which can return a BOOL result or a error code in a callback function.
FFS_SUCCESS	Operation successful
FFS_INITIALIZING	FFS is currently performing initialization, and operations are not allowed.
FFS_ALREADY_OPEN	The requested file is already open, and cannot be accessed.
FFS_NOT_OPEN	The requested file is not open, and operations are not allowed.
FFS_FILE_NOT_FOUND	The specified file doesn't exist.
FFS_EXISTS (FFS_FILE_ALREADY_CREATED)	The file or directory already exists.
FFS_ILLIGAL_ID	The Id specified is not allowed (outside range), or the Id is already used.
FFS_ILLIGAL_FILE_HANDLE	The specified Id doesn't match the specified file handle.
FFS_ILLIGAL_TYPE	The file type specified doesn't exist.
FFS_ILLIGAL_MODE	The mode specified doesn't exist.
FFS_FILE_RANGE_ERROR	Read / write attempted outside the range of the file.
FFS_OPERATION_NOT_POSSIBLE	The requested operation is not possible because the user already have an active operation.
FFS_WRITE_ERROR	Write attempts have been made to already written elements in the file.
FFS_USER_ID_ERROR	The ID used is either illegal or already in use by another FFS operation.
FFS_INTERNAL_FATAL_ERROR	The RAM based file management tables have been corrupted, and the FFS operations are disabled until a power off/on has been made.
FFS_MEMORY_RES_ERROR	The memory reservations exceeds the available memory space.
FFS_MAX_NOF_FILES_EXCEEDED	The maximum number of files in the FFS is reached. Files most be deleted before new files can be created.
FFS_REQUESTED_MEM_NOT_AVAILABLE	The requested amount of memory is not available.
FFS_INVALID_NAME (FFS_INVALID_FILENAME)	The filename or directory path contains invalid characters
FFS_STREAMING_NOT_ENABLED	The read and write functions was called with the predefined offset indicating streaming read/write, but streaming data isn't included in FFS
FFS_OPERATION_NOT_ALLOWED_ON_STATIC_FILE	It is not allowed to append, modify, delete or rename a static file. Neither is it allowed to rename the dynamic copy of a factory file, since this requires changes in the static file.

FFS_MEM_TABLES_INCONSISTENCY	One or more of the internal memory tables doesn't match the security copy. Restart to fix it.
FFS_NOT_A_FACTORY_DEFAULT_FILE	Trying to restore a file which isn't a part of a factory link.
FFS_REQUESTED_MEM_TEMP_NOT_AVAILABLE	There is currently not enough free space to do the requested operation, but when the garbage collector has cleaned up there will be enough free space.
FFS_ILLIGAL_DIR_OPERATION	E.g. when deleting directories containing files.
FFS_DIR_SPACE_NOT_AVAILABLE	The max. number of directories has been reached.

References

[1]	Title:	EEPROM specification
	Author:	Erik Christensen
	Revision:	0.8