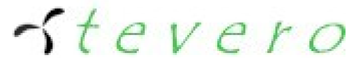


Transaction Flash FileSystem (TFFS) Description

Copyright Tevero AS

This document may be freely distributed as long as it is not changed without the consent of Tevero AS. The mechanisms described within the document shall not be used in any form of device or application without the consent of Tevero AS.

Table of Contents

1 Introduction......4

1.1 Scope......4

1.2 Background......4

1.3 Overview......4

1.4 Implemented File Functionality......5

1.4.1 Basic......5
1.4.2 Access Attributes......5
1.4.3 Protective Mode......7
1.4.4 Access Monitor......7
1.4.5 Timestamp......7

1.5 Disk Usage Saturation......8

1.6 Write Performance Considerations......8

1.7 TFFS Versions Support......8

2 Application Programming Interfaces (APIs)......10

2.1 The TFFS Protection Application Programming Interface (TP-API)......10

2.1.1 TFFS Usage Protection......10
2.1.1.1 Pseudo Code Example......10
2.1.1.2 Posix Code Example......11
2.1.1.3 sig_semRead......12
2.1.1.4 sig_semWrite......12
2.1.1.5 wai_semRead......12
2.1.1.6 wai_semWrite......12
2.1.2 Protective Mode Application Programming Interface......12
2.1.3 Monitor Protection Application Programming Interface......13

2.2 The TFFS User Application Programming Interface (TU-API)......14

2.2.1 tffsChange......14
2.2.2 tffsChdir......15
2.2.3 tffsClearerr......15
2.2.4 tffsDelete......16
2.2.5 tffsDiagMode......16
2.2.6 tffsExists......17
2.2.7 tffsFclose......17
2.2.8 tffsFeof......18
2.2.9 tffsFerror......18
2.2.10 tffsFflush......18
2.2.11 tffsFopen......19
2.2.12 tffsFormat......21
2.2.13 tffsFread......22
2.2.14 tffsFseek......22
2.2.15 tffsFwrite......23
2.2.16 tffsGetOptimalBuffer......23
2.2.17 tffsGetBlockInfo......24
2.2.18 tffsGetPath......24
2.2.19 tffsGetRoot......25
2.2.20 tffsInit......25
2.2.21 tffsLsdir......26
2.2.22 tffsMkdir......27
2.2.23 tffsMonitorAdd......28
2.2.24 tffsMonitorRemove......30
2.2.25 tffsMount......31

2.2.26 tffsReclaim....32
2.2.27 tffsResize....33
2.2.28 tffsRewind....33
2.2.29 tffsTimestampSet....34
2.2.30 tffsTimestampUpdate....34
2.2.31 tffsTruncate....35
2.2.32 tffsUnmount....35

Appendix A: TFFS Integration Guide....36

1 Introduction

1.1 Scope

The current scope of this document is to describe the overview and use of the transaction flash filesystem (TFFS). Please refer to "***Transaction Flash FileSystem (TFFS) Internals***" document (available with product only) for detailed internal description.

Notice that it is strongly recommended to use the available File Abstraction Layer (FAL) wrapper, providing an almost ANSI/ISO compliant file interface in most cases. Then, this document is normally only needed during integration and for special use.

There exists two different versions of TFFS disk layout that are supported, version 1 and 2. Please refer to **1.7 TFFS Versions** for details on restrictions with respect to supporting the two versions.

1.2 Background

Why another filesystem? Well, we were in need of a flash-based filesystem for smaller embedded systems that

- had a small footprint
- could endure power losses and other unexpected restarts without corrupting the file system
- fast mounting during startup
- quick and efficient
- no royalties and cheap

TFFS is not the greatest filesystem of all times, but it solved the above points. At least we think so...

- The footprint is appr. 24KB (PowerPC) including debugging functionality
- Mount time is constant and independent of number of files and directories (assuming recovery was not required)
- TFFS uses transactions in order to complete or rewind unfinished transactions at start-up. This avoids more time consuming consistency checking at start-up.
- It is specially made to work with FDC (the flash disk controller product); we believe it makes the most out of it.
- RDC (RAM disk controller) has been added later and can be used in order to use TFFS on RAM media (both temporary and for instance battery backed up RAM). RDC avoids much of the overhead needed for FDC, yet fulfills the requirements by TFFS in order to handle recoverability (which is needed on non-volatile RAM partitions).
- In addition, it is possible to use TFFS with a 3rd party controller, if a suitable controller adaptation exists.

The main goal for TFFS, is that power can be switched off at any point, without corrupting the filesystem.

TFFS is mainly suitable for smaller systems, with disks sizes ranging from less than 1MB to a few 100MB. In theory, disks up to 4GB is supported.

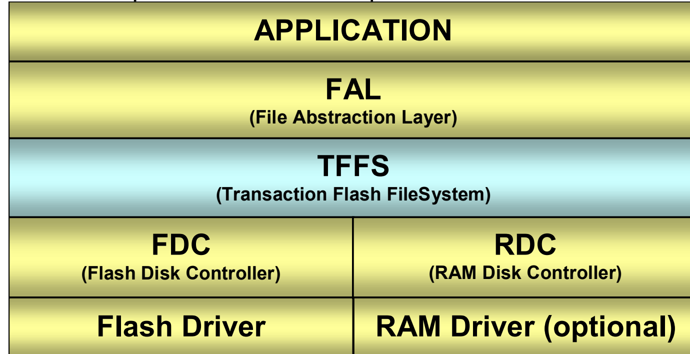
This document in general has a focus on flash devices, since this requires more advanced functionality than when using RAM media.

1.3 Overview

The transaction flash filesystem (TFFS) is a small filesystem specially made for use with the flash disk controller (FDC) and RAM disk controller (RDC) in an embedded system. (nDC is used in this document to indicate either FDC and/or RDC.) It is fully recoverable, by using a transaction record

scheme for those filesystem actions that changes two or more blocks. The underlying nDC layer is able to ensure consistent storage of one single block.

TFFS is a part of a modular implementation, as shown below:



TFFS supports files and directories with attributes (read/write/hidden) but no user identification. A user can have one of three access modes: normal/hidden/super.

The transaction itself is stored on the media used. Since nDC is fully recoverable on its operations, it makes the overall filesystem recoverable from power failures and system crashes by rewinding or completing transactions. By using transaction records, full disk scans are not required during startups, saving startup time, an important issue in many embedded systems.

Although TFFS may be used standalone with nDC, it is recommended to use the file abstraction layer (FAL) layer available or another abstraction layer in order to make application SW as standard as possible. The existing FAL should be sufficient in most cases.

1.4 Implemented File Functionality

1.4.1 Basic

TFFS implements the following filesystem functionality:

- Directories can be created/deleted/renamed (they must be empty to be deleted)
- A file can be created/deleted/renamed
- A file can be read at random locations
- A file can be overwritten at random places
- A file can have data appended to it
- File/directory names can be max 28 characters
- A file/directory can have read/write/hidden attributes
- A file can be truncated
- Optional support for ‘errno’ updating
- Optional support for monitoring actions through the use of callbacks
- Optional support for protective mode which will ensure exclusive write access to a file and inhibit a file in use to be deleted
- Optional support for timestamping (TFFS version 2 only)

ASCII translation mode (CR -> CRLF) is not supported.

1.4.2 Access Attributes

TFFS includes access attributes for files and directories. This may typically be useful in embedded systems that include FTP, TELNET, HTTP support etc. through the use of the TFFS filesystem. Since FTP for instance should allow a user to list directories, retrieve files, delete files etc., it causes

a problem if the system has files a user should not see or delete; for instance system SW files, key files etc.

By using access attributes, the system can hide directories and/or files from an FTP client (or other user). A user can have three different access rights:

NORMAL – This gives normal access rights. The user follows write/read restrictions and can not work on or see hidden files/directories.

HIDDEN – The user follows write/read restrictions, but can work on and see hidden files/directories.

SUPER – The user can do anything. Typically used by the system. **Notice:** This mode overrides protect mode if enabled, and can be destructive to disk if used wrongly, for instance if modifying files already in write use by other parts of the system!

Notice that created files/directories gets its attributes as specified when created. There is no implicit inheritance from the parent directory. A file or directory can be created without any attributes.

The behavior of the attributes for files and directories are somewhat different depending whether it is a file or a directory. Below is shown their meaning in directories for NORMAL/HIDDEN access.

Directory attributes	
Read	The files in the directory may be listed.
Write	New entries may be created in and deleted from directory. Attributes and/or name for all entries in the directory may be changed.
Hidden	The user must have HIDDEN access rights in order to list files in the directory, create new entries or to modify attributes/name for entries (as specified by the read/write attributes of the directory).
No attributes set	Directory can only be accessed with SUPER access rights.

File attributes	
Read	The file may be read.
Write	The file can be written to.
Hidden	The user must have HIDDEN access rights in order to read or write to the file (as specified with the read/write attributes of the file).
No attributes set	File can only be accessed with SUPER access rights.

When accessing a file, it is the attributes for the file, not its parent directory that is of importance. Thus a file with read/write access in a directory with hidden access, can be read and modified by a user with NORMAL access rights as long as the file itself is not hidden. Also, if a file is read only, and the intention is that no user shall write to it, the parent directory should be set to read only, since otherwise anyone can change the file attribute to read/write (as long as changing attributes is possible for a user).

From the above, it follows that if the directory “cd” in “/cd/ef” has hidden access, but “ef” not, the user with NORMAL access rights will not see any entries when trying to list “/cd”, in fact trying to list “/cd” will fail. However, the user can still access “/cd/ef”.

Notice that in order to change attributes and/or name for an entry, the parent directory must have write access (if the parent directory has hidden attribute, the user must have HIDDEN access).

1.4.3 Protective Mode

TFFS includes an optional protective mode, enabled by default (see TFFS_PROTECT define in *tffsconf.h*). This makes it possible to ensure that two users do not write simultaneously to the same file in a multi-user environment, or that one user deletes a file being accessed by another user. In other words, in this mode a file can only be opened once in write access mode, but multiple times in read only mode.

This mode is typically useful in systems where file access is not controlled by the application, such as by FTP clients etc.

Notice that a user with SUPER access rights is not restricted by the above rules.

Obviously, it is very important to close files in this mode when finished, since the files otherwise may be left inaccessible.

When the protective mode feature is enabled, an additional protection mechanism must be added in order to allow a linked list to be accessed safely in a preemptive environment. Please refer to chapter ***2.1.2 Protective Mode Application Programming Interface***.

1.4.4 Access Monitor

TFFS includes an optional access monitor. The monitor can be used to register callback functions on certain filesystem operations. For instance if a parameter file is updated via FTP, the parameters can be processed by the system after the file has been closed, ensuring that the parameter changes take effect.

The following actions can be monitored (actions can be ORed together for an item):

- opening a file with read access
- opening a file with write access
- opening a file with append access
- closing a file with read access
- closing a file with write access
- closing a file with append access
- deleting a file/directory
- renaming a file/directory from a name
- renaming a file/directory to a name

The renaming feature can be useful to detect if a file is stored as an unmonitored file and then renamed into a monitored file, again something that can be done with for instance FTP access. It can also be used to detect if a file is renamed from a monitored item to another name, in effect the same as deleting the monitored item.

When the monitor feature is enabled, an additional protection mechanism must be added in order to allow a linked list to be accessed safely in a preemptive environment. Please refer to chapter ***2.1.3 Monitor Protection Application Programming Interface***.

1.4.5 Timestamp

As of TFFS version 2 layout and later, file and directory timestamps are supported if enabled (see TFFS_TIMESTAMP define in *tffsconf.h*). There are certain limitations that should be considered:

- For directories, only create time is stored. Modified timestamp for a directory is only set at creation time, and thus equal to create time.

- For files, both create and modified timestamps are maintained. The create and modified timestamps are set when a file is created. The create timestamp will not be changed during the lifetime of a file; the file must be deleted and recreated in order to update it. Notice that opening a file in erase mode (e.g. fopen(“foo.txt, “wb+”)) will erase the file before accessing the file, but not delete the file. Thus, the original create timestamp is maintained.

  The modified time is only updated when closing an opened file and if the file has been modified since last update of the modified time. In addition, the modified time may be explicitly set for an opened file with *tffsUpdateMtime()* from the application. This restriction is set in order to reduce the requirements on file update, i.e. a performance issue.

Updating the modified timestamp requires the directory where the file is located to be search. This search time is on average proportional to number of entries in a directory.

1.5 Disk Usage Saturation

Notice that when using flash disks a problem arises when the disk fills up. In short, the fuller the disk is, the less number of blocks on average are recovered whenever erasing a flash erase zone. Since erase operations are very time consuming (worst case up to several seconds per erase zone, it depends on the flash device), the disk write performance degrades when the reclaiming frequency increases. Since TFFS also needs to add some file system info (block allocation table), this causes extra block writes and an escalation of the problem when the disk approaches full.

There is no good remedy for this, except dimensioning the disk to be big enough for normal usage.

Notice that this is not a problem for RAM disks.

1.6 Write Performance Considerations

TFFS allocates a buffer with a fixed size of one block of the underlying media for every file opened. This is done in order to be able to buffer small changes and thus reducing the number of write operations to the media. This is much the same was as buffered IO works with ISO C fwrite().

The most common write operation is when writing sequentially to a file, appending blocks. Appending a block to a file requires the use of a transaction since the block allocation table must also be updated. Every transaction in itself requires some overhead.

By passing larger buffers to tffsFwrite(), TFFS is able to append several blocks in one transaction. This can increase the write performance in some cases considerably, since less overhead is needed on average per appended block.

The maximum number of bytes that TFFS can write in one transaction can be fetched with tffsGetOptimalBuffer(). Although it can be used to achieve optimal write performance, we do not recommend to rely heavily on such a non-standard feature in application SW. Quite a bit of performance gain can be achieved by just providing tffsFwrite() with buffers of 3-4 blocks.

The ISO C standard library supports the possibility of configuring the internal stream buffer with a setvbuf() function. TFFS does currently not support that feature for various reasons.

1.7 TFFS Versions Support

There exist currently two different layout formats of TFFS disks, version 1 and 2. Version 2 layout adds support for create and modify timestamps with certain limitations. TFFS version 2 SW (3.x.x) can mount and operate on TFFS version 1 formatted disks. However, since transaction sequencing

has changed for some transaction types, TFFS version 2 SW will not be able to handle all transactions created by TFFS version 1 SW (2.x.x.) properly in all cases, and vice versa.

For the above reason, it is important to understand that a switch (upgrade or downgrade) between TFFS version 1 and 2 SW must only be done in a controlled manner, i.e. there should not be a pending transaction to be recovered when the new SW is started. A pending transaction may only be present when mounting a disk, if the system restarted uncontrolled at “the wrong” place during a disk write operation.

TFFS version 1 SW (2.x.x) cannot mount TFFS version 2 layout formatted disks. By default, TFFS version 2 SW will format new disks to version 2 layout. If a downgrade to TFFS version 1 SW is foreseen, it should be considered if one should change the default format to version 1 (controlled by TFFS_USE_VERSION define).

2 Application Programming Interfaces (APIs)

2.1 The TFFS Protection Application Programming Interface (TP-API)

2.1.1 TFFS Usage Protection

In order to use in a preemptive system, the user must provide protection functions to TFFS if:

- there is no protection on a higher layer
- TFFS is going to be used in more than one context

TFFS allows only one user to write to a partition at a time. A write operation must complete before anyone else can read or write. This also implies that a write operation can not start before all read operations are completed.

Notice that by read/write operations above, we mean operations of a few block lengths. A file read/write, will always be split into a (smaller) number of block accesses, and will allow higher priority threads to access the disk once the lower priority thread has finished its current block access operation.

The reason for only allowing one to write at a time is somewhat reasonable when using flash media. In that case, the rest of a flash device can typically not be read, while writing to it. *(Notice that dual-plane flashes make it possible to read while writing to some extent.)*

When using RAM as media, the reasons are not obvious. However, TFFS uses serialized transactions, and a transaction must be completed before a new one is executed. The limitation with only one writing at a time is not a big problem in most contexts, since each write operation is usually short (only a few blocks worst case), and when finished, higher priority task will get access.

To implement the protection mechanism in an OS environment, it would most likely mean that one has to use a semaphore, an event and a counter to handle the situation.

If Posix read/write locks are available (pthread_rwlock_wrlock() and pthread_rwlock_rdlock()), then this is exactly what is needed.

2.1.1.1 Pseudo Code Example

An implementation suggestion is indicated below with pseudo-code; be careful to use the defined sequence of operations:

```
U32 readersActive = 0;

int wai_semRead(void *userPointer)
{
   /* Wait for possible write to finish. */
   Obtain semaphore

   /* If timeout is supported */
   if (timeout)
      return(-1);

   /* Increment number of readers, this will also inhibit new write */
   /* accesses to take place. We can't rely on semaphore protection here */
   /* since the corresponding releasing read access function does not */
   /* use semaphores. */
   disable preemption
   readersActive++;
   enable preemption

   Release semaphore

   return(0);
```

```
}

void sig_semRead(void *userPointer)
{
   /* Decrement number of readers. Can't use semaphore here */
   /* since a user waiting for write may have taken it. */
   disable preemption
   if (readersActive)
      readersActive--;
   enable preemption

   /* If this was the last one reading, set event in case someone */
   /* is waiting for write access. */
   if (!readersActive)
      set "last read completed" event
}

int wai_semWrite(void *userPointer)
{
   /* Get exclusive access to flash, inhibiting new read accesses. */
   Obtain semaphore

   /* If timeout is supported */
   if (timeout)
      return(-1);

   /* Clear read event in case it has been set earlier */
   clear "last read completed" event

   /* If anyone is reading, wait for last reader to finish */
   if (readersActive)
   {
      wait for "last read completed" event

      /* If timeout is supported */
      if (timeout)
      {
         Release semaphore
         return(-1);
      }
   }

   return(0);
}

void sig_semWrite(void *userPointer)
{
   /* Let others request access */
   Release semaphore
}
```

2.1.1.2 Posix Code Example

```
static pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

int wai_semRead(void *userPointer)
{
   /* Wait for possible write to finish. */
   if (pthread_rwlock_rdlock(&rwlock) < 0)
      return(-1);
}

void sig_semRead(void *userPointer)
{
   /* Release lock */
   pthread_rwlock_unlock(&rwlock);
}

int wai_semWrite(void *userPointer)
{
   /* Wait for all read operations to finish. */
   if (pthread_rwlock_wrlock(&rwlock) < 0)
      return(-1);
```

```
}

void sig_semWrite(void *userPointer)
{
   /* Release lock */
   pthread_rwlock_unlock(&rwlock);
}
```

2.1.1.3 sig_semRead

Prototype:	***void sig_semRead(void *userPointer)***	
Description:	End read access to the flash array.	
Reentrancy:	Reentrant	
Input:	*userPointer*	Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

2.1.1.4 sig_semWrite

Prototype:	***void sig_semWrite(void *userPointer)***	
Description:	End writes access to the flash array.	
Reentrancy:	Reentrant	
Input:	*userPointer*	Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.

2.1.1.5 wai_semRead

Prototype:	***int wai_semRead(void *userPointer)***	
Description:	Wait for read access to the flash array. Multiple users may access the flash devices as long as no one is writing to it.	
Reentrancy:	Reentrant	
Input:	*userPointer*	Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.
Output:	*int*	0 - access granted <0 – unable to grant access (operation cancelled)

2.1.1.6 wai_semWrite

Prototype:	***int wai_semWrite(void *userPointer)***	
Description:	Wait for write access to the flash array. Only one may write to the flash array at a time, and none may read at the same time.	
Reentrancy:	Reentrant	
Input:	*userPointer*	Pointer provided by user and passed from FDC. Can be used to identify driver type to be used, partition etc.
Output:	*int*	0 - access granted <0 – unable to grant access (operation cancelled)

2.1.2 Protective Mode Application Programming Interface

If protective mode is enabled (see***1.4.3 Protective Mode***), then an almost identical set of protection functions, as described in ***2.1 The TFFS Protection Application Programming Interface (TP-API),*** must be provided in order to protect the monitor list. The only difference is that both input and output parameters are void. The protective mode protection API must operate independently of the general TP_API, otherwise deadlocks may occur.

The provided protection API must be defined in *tffsconf.h*.

2.1.3 Monitor Protection Application Programming Interface

If access monitoring is enabled (see ***1.4.4 Access Monitor***), then an almost identical set of protection functions, as described in ***2.1 The TFFS Protection Application Programming Interface (TP-API),*** must be provided in order to protect the monitor list. The only difference is that both input and output parameters are void. The monitor protection API must operate independently of the general TP_API, otherwise deadlocks may occur.

The provided protection API must be defined in *tffsconf.h*.

2.2 The TFFS User Application Programming Interface (TU-API)

The user functions are the functions available for user of TFFS. A typical user would be a file abstraction layer (FAL) and to some extent the application itself.

2.2.1 tffsChange

```
int tffsChange(
    int id,
    U32 currentDir,
    const char *item,
    U8 *attrib,
    const char *name)
```

Description: Change attributes/name for directory or file.

Reentrancy: Reentrant

Input:	*id*	ID for disk to operate on
	currentDir	Block number holding start of current active directory
	item	File or directory to modify (may include relative path)
	attrib	Ptr to new attributes for file or directory. Use NULL if attributes shall not be changed. If provided, it should point to logical OR of zero or more of the following values: TFFS_READ, TFFS_WRITE, TFFS_HIDDEN
	name	New name for file or directory. Use NULL if name shall not be changed.
Output:	*int*	0 - Changed <0 - Not changed

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EEXIST, EBUSY, ENOENT, EPERM, EACCES or EIO

2.2.2 tffsChdir

int tffsChdir(int id, U32 currentDir, const char *dir, U32 *newDir)

Description:	Change directory.	
Reentrancy:	Reentrant	
Input:	*id*	ID for disk to operate on
	currentDir	Block number holding start of current active directory
	dir	Path to new directory (relative or absolute)
	newDir	Location to place block number for start of new directory
Output:	*int*	0 - Changed <0 - Not changed (TFFS_ERROR_NONEXISTING) or other fault

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, EIO, EACCES or ENOTDIR

2.2.3 tffsClearerr

void tffsClearerr(TFFS_FILE *stream)

Description:	Clear error indicator for a stream. The error indicator for a stream is not automatically cleared, and can only be cleared with tffsClearerr(), tffsRewind() or tffsFclose(). EOF is cleared if set.	
Reentrancy:	Reentrant	
Input:	*stream*	Pointer to file structure for file to clear error

2.2.4 tffsDelete

int tffsDelete(int id, U32 currentDir, const char *item)

Description: Delete a file or directory. If deleting a directory, the directory must be empty in order to delete it.

WARNING:
TFFS does not inhibit deleting a file that is opened unless protection is enabled (TFFS_PROTECT define). Notice that a SUPER user can always delete a file, even if protection is enabled.

Reentrancy: Reentrant

Input:

Input:	*id*	ID for disk to operate on
	currentDir	Block number to current active directory.
	item	Item to delete (including possible relative path to currentDir).
Output:	*int*	0 - Deleted <0 - Not deleted

If 'errno' usage is enabled, 'errno' is updated on error to: NXIO, EINVAL, EBUSY, EIO, ENOENT, EACCES, ENOTEMPTY or EPERM

2.2.5 tffsDiagMode

int tffsDiagMode(int id, U8 mode)

Description: Control diagnostic mode for a disk.

Reentrancy: Reentrant

Input:	*id*	Disk ID for disk to set diagnostic mode
	mode	Diagnostic mode to set. Use one of TFFS_DIAG_DISABLE - Disable diagnostics logging TFFS_DIAG_ERRORS - Show errors TFFS_DIAG_ALL - Show all diagnostics actions and errors
Output:	*int*	0 if no error has occurred. Non-zero value if error has occurred.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL or EBUSY

2.2.6 tffsExists

int tffsExists(int id, U32 currentDir, const char *item, TFFS_STAT *stat)

Description: Check if file or directory exists and get its status

Reentrancy: Reentrant

Input:

id	ID for disk to operate on
currentDir	Block number the start of the directory file of the current directory.
item	Item to check if exists.
stat	Ptr to location to store status for item if found. Use NULL if status not wanted.

Output:

int	0 - Item does not exist (or is hidden) !0 - Item exists

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL or EBUSY

NOTICE: 'errno' is only updated due to an actual usage or internal fault. The fact that an item does not exist is not considered a fault.

2.2.7 tffsFclose

int tffsFclose(TFFS_FILE *stream)

Description: Close a file. The buffer is flushed before closing, if required.

Reentrancy: Reentrant

Input:

stream	Pointer to TFFS file stream to close
mode	Diagnostic mode to set. Use one of TFFS_DIAG_DISABLE - Disable diagnostics logging TFFS_DIAG_ERRORS - Show errors TFFS_DIAG_ALL - Show all diagnostics actions and errors

Output:

int	0 if file closed successfully. TFFS_EOF is returned if fault encountered.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EBUSY, EIO or ENOSPC

2.2.8 tffsFeof

int tffsFeof(TFFS_FILE *stream)

Description: Check if end of file reached.

Reentrancy: Reentrant

Input: *stream* Pointer to file structure for file to check for EOF

Output: *int* 0 if EOF not reached. TFFS_EOF is returned if end reached.

2.2.9 tffsFerror

int tffsFerror(TFFS_FILE *stream)

Description: Test for an error on a stream. Notice that when an error has occurred for a stream, it must be explicitly cleared by tffsClearerr(), tffsRewind() or tffsFclose().

Reentrancy: Reentrant

Input: *stream* Pointer to file structure for file to check for error

Output: *int* 0 if no error has occurred.
Non-zero value if error has occurred.

2.2.10 tffsFflush

int tffsFflush(TFFS_FILE *stream)

Description: Flush buffers if required. (Only if buffers are modified.)

Reentrancy: Reentrant

Input: *stream* Pointer to TFFS file stream to flush.

NOTICE: Does not support use of NULL pointer which according to ANSI/ISO C should flush all streams.

Output: *int* 0 if file flushed successfully. TFFS_EOF is returned if fault encountered.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EBUSY, EIO or ENOSPC

2.2.11 tffsFopen

TFFS_FILE *tffsFopen(
int id,
U32 currentDir,
U8 attrib,
const char *filename,
const char *mode)

Description: Flush buffers if required. (Only dirty buffers are flushed.)

Reentrancy: Reentrant

Input:

id — ID for disk to operate on

currentDir — Block number for start of current directory file

attrib — Attributes for file if file is created. Logical OR of zero or more of: TFFS_READ, TFFS_WRITE, TFFS_HIDDEN

filename — Name of file (including possible path) to open

mode — Mode to open file in. The following modes are supported:

(TFFS does currently not support the text mode for files, i.e. translation of LF to CRLF; everything is treated as binary. It is recommended to use the intended mode.)

"ab", "a"
Open for writing only in binary mode at the end (appending). File is created if it does not exist.

"ab+", "a+", "a+b"
Open for reading and appending in binary mode. File is created if it does not exist. File can be read at any location, but when writing, the file pointer is always moved to the end first.

"rb", "r"
Open for read only in binary mode. If file does not exist, the operation fails.

"rb+", "r+", "r+b"
Open for update (read/write) in binary mode. If file does not exist, the operation fails. This mode allows for actually updating the contents of an existing file.

"wb", "w"
Open for write only in binary mode. File is created if it does not exist, and erased if it exists.

"wb+", "w+", "w+b"
Open for read/write in binary mode. File is created if it does not exist, and erased if it exists.

The mode string may contain more characters than the above listed sequences (according to ANSI/ISO C); they are ignored by this implementation.

Output: *TFFS_FILE ** — Pointer to FILE structure for stream. NULL if operation could not be completed.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, ENOMEM, EBUSY, EIO, EISDIR, ENOENT, EACCES, ENOSPC

2.2.12 tffsFormat

```
int tffsFormat(
    int prtId,
    U32 maxBlocks,
    TFFS_DISK_CTRL *dapi,
    void *userPointer,
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 diagMode)
```

Description: Format a disk for TFFS usage. The underlying partition should be in a clean formatted state before formatting for TFFS usage.

NOTICE:
This function allocates memory needed for various issues. The memory is kept only during formatting.

Reentrancy: Reentrant

Input:

	prtId	Handle for partition to operate on.
	maxBlocks	If the disk is to be expanded in the future, this must be planned ahead at format time in order to reserve space for the BAT entries. Set this to 0 to only make BAT as large as needed for the current sized partition. If the partition shall be expanded at some time, then set maxBlocks to the number of blocks (denoted logical blocks in partition) that will be available.
	dapi	Pointer to structure that contains disk API info. Use proper init function available with the disk controller API. The structure is not required after finishing formatting.
	userPointer	Pointer that can be set by the user and possibly used by user provided functions. Not used by TFFS.
	wai_semWrite	Function pointer to function waiting for write access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
	sig_semWrite	Function pointer to function releasing the write access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
	diagMode	Diagnostic mode to use while formatting. Use TFFS_DIAG_xxx defines. Only used if TFFS_DIAG is enabled.

Output:

	int	0 - Format completed. <0 - Failure in formatting

If 'errno' usage is enabled, 'errno' is updated on error to: EINVAL, ENOMEM, EBUSY, EIO, ENOSPC

2.2.13 tffsFread

U32 tffsFread(void *buffer, U32 size, U32 count, TFFS_FILE *stream)

Description: Read from an opened file.

Reentrancy: Reentrant

Input:	*buffer*	Pointer to buffer to place read data
	size	Size in bytes of an element to read
	count	Number of elements to read
	stream	Pointer to TFFS file stream to read from
Output:	*U32*	Number of elements read. If less than specified and EOF reached, the EOF flag for the stream is set.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EACCES, EIO, EBUSY

2.2.14 tffsFseek

int tffsFseek(TFFS_FILE *stream, long offset, int origin)

Description: Set position in file. The new position will be used for the next read or write.

If stream buffer has been modified, the buffer will be flushed according to ANSI/ISO C definition.

If seeking beyond the end of file, the next write (if write access allowed and not append mode) will cause the gap to be filled with 0s. A read will cause an EOF condition.

Reentrancy: Reentrant

Input:	*stream*	Ptr to TFFS file stream to change position for
	offset	Offset in bytes from origin to move. It is not allowed to seek before start of file (returns with fault). This can be both a positive and negative value.
	origin	Where to seek from: TFFS_SEEK_SET - Start of file TFFS_SEEK_CUR - Current position of file pointer TFFS_SEEK_END - End of file
Output:	*int*	0 - Successful operation, EOF is cleared. !0 - Operation failed (typically negative error code). File pointer not moved.

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, ENOSPC, EIO

2.2.15 tffsFwrite

U32 tffsFwrite(const void *buffer, U32 size, U32 count, TFFS_FILE *stream)

Description: Write to an opened file.

NOTICE: Providing a larger buffer may increase write performance when appending data to a file due to including multiple blocks in a single transaction. This reduces the average transaction overhead. The max buffer that can be added in one transaction can be obtained with tffsGetOptimalBuffer(). Exceeding that buffer size does not yield any performance gain, but is of course allowed.

Reentrancy: Reentrant

Input:	*buffer*	Pointer to buffer that holds data to be written
	size	Size in bytes of an element to write
	count	Number of elements to write
	stream	Pointer to TFFS file stream to write to
Output:	*U32*	Number of elements written

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, ENOSPC, EIO

2.2.16 tffsGetOptimalBuffer

U32 tffsGetOptimalBuffer(TFFS_FILE *stream)

Description: Get optimal buffer for fully utilizing multiple blocks in a transaction when writing to stream.

When appending data to a file, including multiple blocks in a single transaction will increase write performance. This is not the same as the ISO C setvbuf() feature, which can be used to increase a stream buffer for increased performance. TFFS has a fixed stream buffer of one block, but when providing a larger amount of user data, several blocks can pass through TFFS and be written within one transaction. The optimal buffer size passed to tffsFwrite (ie the maximum number of bytes that can be written in one transaction) depends on the block size and implementation internals.

Although it is not encouraged to heavily depend on this non-standard feature, it may be useful in cases where write performance is important.

Notice that as a "rule of thumb", the main performance gain is achieved if providing appr. half the size of the returned value. However, in order to get the optimal gain, the returned buffer size should be used.

Reentrancy: Reentrant

Input:	*stream*	Pointer to TFFS file stream to get optimal buffer size for
Output:	*U32*	Optimal buffer size in number of bytes

2.2.17 tffsGetBlockInfo

int tffsGetBlockInfo(int id, U32 *totalBlocks, U32 *usedBlocks, U32 *blockSize)

Description: Get block information for disk

Reentrancy: Reentrant

Input:	*id*	Disk ID for disk to get info
	totalBlocks	Ptr to location to place number of usable blocks on disk. Use NULL if not requesting this info.
	usedBlocks	Ptr to location to place number of used blocks on disk. Use NULL if not requesting this info.
	blockSize	Ptr to location to place block size (in bytes) used on this disk. Use NULL if not requesting this info.
Output:	*int*	0 - OK, info fetched <0 - Invalid disk ID

2.2.18 tffsGetPath

int tffsGetPath(int id, U32 dir, char *path, U32 max)

Description: Get the path for a directory

Reentrancy: Reentrant

Input:	*id*	ID for disk to operate on
	dir	Block holding directory to get path for.
	path	Ptr to location to place path. The path is terminated by 0x00.
	max	Max length of path string allowed. This is used in order to inhibit writing outside the 'path' buffer. Notice that the 'path' buffer must be one more than max, in order to hold 0x00 termination.
Output:	*int*	0 - Path fetched <0 - Not fetched path

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EIO or EBUSY

2.2.19 tffsGetRoot

U32 tffsGetRoot(int id)

Description: Get the block number for the root directory in a TFFS disk.

Reentrancy: Reentrant

Input:	*id*	ID for disk to operate on
Output:	*U32*	0 - Error >0 - Block number for root directory

2.2.20 tffsInit

void tffsInit(U32 max)

Description: Initialize TFFS system. Should be invoked once before using TFFS.

Reentrancy: Reentrant (well, almost...)

Input:	*max*	Max number of TFFS disks that can be defined. A pointer is allocated for every disk that can be defined. The pointer table is later used for fast lookup.

2.2.21 tffsLsdir

```
int tffsLsdir(
    int id,
    U32 currentDir,
    const char *dir,
    U32 entryNo,
    Char *name,
    TFFS_STAT *stat)
```

Description: List directory entries.

Reentrancy: Reentrant

NOTICE: If files are removed from a directory during listing, it may cause incomplete listing. The deficiency is caused by the fact that deleted files may cause that he last file in the directory listing is moved to the generated "hole".

Input:	*id*	ID for disk to operate on
	currentDir	Block number to the start of the directory file of the current directory.
	dir	Directory to list files for (if NULL then the current directory is chosen).
	entryNo	Entry element number in directory to list. Entry 0 id "This Dir", entry 1 is "Parent Dir" (these two are always present), then followed by normal entries. Notice that hidden entries are not included in count if user does not have hidden access.
	name	Ptr to location to place file/directory name. The array should be TFFS_FILENAME_MAX + 1 long.
	stat	Returns the status attributes of the file/directory
Output:	*int*	0 - Retrieved <0 - No more entries (TFFS_ERROR_NONEXISTING) or other fault

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EBUSY, EIO, ENOENT, EACCES, ENOTDIR or EINVAL

2.2.22 tffsMkdir

int tffsMkdir(int id, U32 currentDir, const char *dir, U8 attrib)

Description: Make a new directory

Reentrancy: Reentrant

Input:

id	ID for disk to operate on
currentDir	Block holding current directory
dir	Directory to create. Notice that it may be a relative (to currentDir) path or absolute path, but the parent directory of the directory to create must exist.
attrib	Access codes for directory (can be ORed together): TFFS_READ, TFFS_WRITE, TFFS_HIDDEN

Output:

int	0 - Directory created <0 - Not created

If 'errno' usage is enabled, 'errno' is updated on error to: NXIO, EINVAL, EIO, EEXIST, EACCES, ENOSPC, EBUSY

2.2.23 tffsMonitorAdd

```
void *tffsMonitorAdd(
    int id,
    const char *item,
    unsigned long actions,
    void (*callback)(void *callbackPtr, unsigned long action),
    void *callbackPtr)
```

Description: Add monitor action entry on a filesystem item. The same item can be monitored multiple times. Notice that files already opened will not be monitored by the added entry.

NOTICE: The monitor list per disk is searched on every item open and delete in order to check if item is monitored. For this reason, the number of files monitored should not be excessively large...

Reentrancy: Reentrant

Input:

id — ID for disk to monitor item in

item — Item to monitor. MUST be absolute path within the disk. The item does not have to exist.

NOTICE: Directories can only be monitored on delete and/or renaming actions.

actions — Actions to monitor for. This is ORed combination of the following elements:

TFFS_MON_OPEN_READ - opening a file with read access
TFFS_MON_OPEN_WRITE - opening a file with write access
TFFS_MON_OPEN_APPEND - opening a file with append access

TFFS_MON_CLOSE_READ - closing a file with read access
TFFS_MON_CLOSE_WRITE - closing a file with write access
TFFS_MON_CLOSE_APPEND - closing a file with append access

TFFS_MON_DELETE - deleting a file/directory
TFFS_MON_RENAME_FROM - renaming a file/directory from item name
TFFS_MON_RENAME_TO - renaming a file/directory to item name

TFFS_MON_OPEN_WRITE | TFFS_MON_CLOSE_WRITE for a file will invoke the provided callback after opening the file for write and then again after closing it.

Notice that the callback is invoked for each matching action. Thus if specifying
TFFS_MON_OPEN_READ | TFFS_MON_OPEN_WRITE
the callback will be invoked twice if opening the file in RW mode.

	callback	Ptr to callback function to be invoked when specified actions occur. (The callback is invoked AFTER the specified action has finished.) The callback will be invoked using 'callbackPtr' and variable indicating the action triggering the callback. The variable is one of the elements listed above.
	callbackPtr	Ptr to use in callback. May be set by user to hold additional info used in callback.
Output:	*void **	Ptr that must be used if removing this monitoring, NULL if unable to add monitoring.

2.2.24 tffsMonitorRemove

void tffsMonitorRemove(void *entry)

Description: Remove monitoring action entry on a filesystem item. Notice that if any streams for this item are opened, the entry will not be removed until the last opened stream has been closed. Monitoring will however be stopped for this entry.

Reentrancy: Reentrant

Input:	*entry*	Ptr returned when creating monitoring entry

2.2.25 tffsMount

```
int tffsMount(
    int prtId,
    TFFS_DISK_CTRL *dapi,
    void *userPointer,
    int (*wai_semRead)(void *userPointer),
    void (*sig_semRead)(void *userPointer),
    int (*wai_semWrite)(void *userPointer),
    void (*sig_semWrite)(void *userPointer),
    U8 (*getAccess)(void *userPointer),
    U8 diagMode)
```

Description: Mount a TFFS disk.

NOTICE:
This function allocates memory needed for various issues. The memory is kept until unmounting the disk.

Reentrancy: Reentrant

Input:

prtId	Handle for partition to operate on.
dapi	Pointer to structure that contains disk API info. Use proper init function available with the disk controller API. The structure is not required after finishing mounting.
userPointer	Pointer that can be set by the user and possibly used by user provided functions. Not used by TFFS.
wai_semRead	Function pointer to function waiting for read access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
sig_semRead	Function pointer to function releasing the read access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
wai_semWrite	Function pointer to function waiting for write access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
sig_semWrite	Function pointer to function releasing the write access semaphore for the disk. Set to NULL if the disk access shall be treated without preemption concerns.
getAccess	Function pointer to function used to get the access capability for a user.
diagMode	Diagnostic mode to use for disk. Use TFFS_DIAG_xxx defines. Only used if TFFS_DIAG is enabled.

Output:

int	0 - Format completed. <0 - Failure in formatting

If 'errno' usage is enabled, 'errno' is updated on error to: EINVAL, EBUSY, ENOMEM, EIO, ENODEV

2.2.26 tffsReclaim

int tffsReclaim(int id, U8 level)

Description: Reclaim obsolete blocks in a partition up to a certain percentage or just one erase zone.

If the underlying partition does not support reclaiming, this function will return immediately.

NOTICE: This function is used rather than the API for the underlying partition in order to use proper reentrancy protection handling. When using TFFS, the underlying partition will normally not have any reentrancy protection, since this is handled by TFFS.

Reentrancy: Reentrant

Input:

id	ID for disk to operate on
level	If MSB is 0: Percent of obsolete blocks to reclaim (0-100). This means that if there are 100 obsolete blocks, and this parameter is 70, the reclaiming will stop once at least 70 blocks have been reclaimed. 0x80: Reclaim just one erase zone (not done if no obsolete blocks exist). Notice that it is not guaranteed that any obsolete blocks are reclaimed, since every now and then erase zones may be "reclaimed" due to wear leveling, even though they do not have any obsolete blocks.

Output:

int	0 - reclaiming completed <0 - some fault occurred

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, ENOSPC or EIO

2.2.27 tffsResize

int tffsResize(int *tffsId, int *prtId, U16 ezones, U8 front, U8 expand)

Description: Used to resize a disk. Since this requires the underlying partition to be resized, this will be done. When the resizing is completed, the disk is unmounted and remounted. Thus, the partition ID and TFFS disk ID may in theory (but not likely) change when resizing.

NOTE:
a) Resizing (expansion) of a disk MUST currently be planned ahead at format time of the original disk, please see tffsFormat() parameters.

b) Only expansion is currently supported.

c) Resizing is currently not recoverable!!! In order to make it recoverable, some extra functionality must be added to partition opening.

Reentrancy: Reentrant

Input:

	tffsId	Pointer to disk ID for disk to resize. If the resize completes, then the disk is unmounted and remounted, thus the disk ID may change.
	prtId	Pointer to partition ID for partition to resize. If the resize completes, then the partition is closed and reopened, thus the partition ID may change.
	ezones	Number of erase zones to expand/reduce partition with, max 2^14 - 1
	front	0 - Resize at end of current partition 1 - Resize at the beginning of current partition
	expand	0 - Reduce (currently not supported) 1 – Expand

Output: *int* 0 - Resizing completed
<0 - Unable to resize

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, ENOMEM, ENOSPC, EIO or ENODEV

2.2.28 tffsRewind

void tffsRewind(TFFS_FILE *stream)

Description: Rewind file to start and clear error indicator for a stream.

EOF is cleared if set.

Reentrancy: Reentrant

Input: *stream* Pointer to file structure for file to rewind

2.2.29 tffsTimestampSet

int tffsTimestampSet(TFFS_FILE *stream, const U32 *ctime, const U32 *mtime)

Description: Set timestamps for an opened file to any value. This function is typically for system usage, please see tffsTimestampUpdate() for just updating modified timestamp. File must be opened with write access.

Reentrancy: Reentrant

Input:

stream — Pointer to TFFS file stream to set timestamp for (must be a valid opened file)

ctime — Ptr to location holding create timestamp to use when setting value. Use NULL if not setting create timestamp.

mtime — Ptr to location holding modified timestamp to use when setting value. Use NULL if not setting modified timestamp.

Output:

int — 0 - Set (if required)
<0 - Unable to set

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EBUSY, ENOENT, EACCES or EIO.

2.2.30 tffsTimestampUpdate

int tffsTimestampUpdate(TFFS_FILE *stream, int force)

Description: Update modified timestamp for an opened file (if file has been modified since the modified timestamp was last updated). File must be opened with write access.

Reentrancy: Reentrant

Input:

stream — Pointer to TFFS file stream to update timestamp for (must be a valid opened file)

force — Set to 1 in order to force updating of modified timestamp even if file has not been modified. Set to 0 for normal use.

Output:

int — 0 - Update done (if required)
<0 - Unable to update

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EBUSY, ENOENT, EACCES or EIO.

2.2.31 tffsTruncate

int tffsFtruncate(TFFS_FILE *stream, U32 length)

Description: Make file have the length specified.

If the file previously was larger than length, the extra data is discarded. If it was previously shorter than length, it is zero- filled.

The file stream position is not changed when using this function, even if reducing the file size to less than the current stream position.

NOTICE:
When making file shorter than it was, a transaction record is made and operation will be completed once started, even if system should be restarted etc. If increasing the file, no transaction is created and the number of zeros padded if a restart should occur is unspecified (but not more than specified). Either way, the file-system integrity is ensured.

Reentrancy: Reentrant

Input: *stream* Ptr to TFFS file stream to truncate. Notice that the file must be opened with WRITE access. The file will be flushed first in case buffer contains modified data.

length Number of bytes file shall have after truncating finished.

Output: *int* 0 - Successful operation
<0 - Operation failed

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO, EINVAL, EBUSY, ENOSPC, EIO, EACCES

2.2.32 tffsUnmount

int tffsUnmount(int id)

Description: Unmount a TFFS disk, e.g. release all used resources.

Reentrancy: Reentrant

Input: *id* Disk ID to unmount

Output: *int* 0 - Unmounting completed.
<0 - Failure in unmounting

If 'errno' usage is enabled, 'errno' is updated on error to: ENXIO or EBUSY

Appendix A: TFFS Integration Guide

This guide assumes that you have a basic knowledge of your compiler and build environment. It provides a step-by-step instruction in order to start using TFFS with FDC on your target. (Notice that using TFFS with RDC is very similar and not shown here.)

1. **FDC Integration:** Make sure that the FDC integration as described in the FDC documentation is completed successfully.

2. **Configuration:** Do the entire configuration required by modifying the *src/tffs/tffsconf.h* file. It is recommended that debugging is enabled at first, if some support for *printf()* is provided. (*src/shared/shrdconf.h* is assumed already configured with the integration of FDC.)

3. **Ensure that TFFS compiles in your environment:** First include the TFFS source files (*src/tffs/*.c*) in your make or build environment. You should ensure that all the include files (*src/tffs/*.h*, *src/shared/*.h* and FDC and/or RDC include files) are available when compiling the source files.

4. **Protection:**

   If TFFS is to be used without higher level reentrancy protection in a multi-user environment, design the protection functions as described in chapter ***2 The TFFS Protection Application Programming Interface (TP-API)***.

   If used with FAL in a multi-threaded environment, this must be done. **Notice** that nDC does not require any reentrancy protection functions when used with TFFS, since it is handled by TFFS. (This assumes no other users of nDC and flash drivers than TFFS.)

   If the protective mode is going to be used (by default enabled, see***1.4.3 Protective Mode***) in a multi-threaded environment, the stream entry list protection API must also be defined. Again, please refer to chapter ***2 The TFFS Protection Application Programming Interface (TP-API)***.

   If the access monitor API is going to be used (see ***1.4.4 Access Monitor***) in a multi-threaded environment, the monitor list protection API must also be defined. Again, please refer to chapter ***2 The TFFS Protection Application Programming Interface (TP-API)***.

5. **Opening and formatting:** Below is a code example on how to open and format an 8MB flash partition and then format it to a TFFS disk. The flash partition consist of 2 4MB devices with 64KB block/sectors (erase zones) designed in serial (thus no interleaving).

   Since this is a bit about fault tolerant SW, one should consider the following possibility: In theory, a FDC partition may be opened and considered valid by FDC, but TFFS fails to use it for mounting. (Something has gone very wrong...) In this case, the partition must be formatted for FDC use before the partition is formatted for TFFS use. Thus, a TFFS mount may fail for two reasons:

   1) The FDC partition has just been formatted, proceed with TFFS format

   2) Something very wrong, the FDC partition must be formatted before proceeding with TFFS format.

```
int applFdcOpen(U8 format, U8 *formatted)
{
   int ret;

   /* Indicate that partition has not been formatted */
   *formatted = 0;
```

```
   /* Should we try opening first (i.e. not force format)? */
   for(;;)
   {
      /* Format if this is requested */
      if (format)
      {
         ret = fdcFormat("EXAMPLE",
                         0,                   /* Disk starts at the beginning */
                         0x800000 / 0x10000, /* Number of erase zones */
                         0,                   /* We don't plan to expand the
                                                 partition in the future */
                         22,                  /* 4MB (2^22) devices */
                         16,                  /* 64KB (2^16) erase zone size */
                         9,                   /* Block size 512 (2^9) bytes */
                         FDC_DEVICES_INTERLEAVED_1 | FDC_DEVICE_BITWIDTH_8,
                         0,                   /* 1 ezone reserved for reclaiming */
                         2,                   /* Additional 3 blocks reserved
                                              /* for reclaiming */
                         NULL,
                         flashErase,
                         flashRead,
                         flashWrite,
                         NULL,                /* No reentrancy protection provided */
                         NULL,
                         FDC_DIAG_ALL);
         if (ret < 0)
         {
            printf("FDC format failed\n");
            return(ret);
         }

         *formatted = 1;
      }

      /* Try to open FDC partition */
      ret = fdcOpen(0,                   /* Start looking from the beginning
                                            of the first flash */
                    0,                   /* Expect to find the partition
                                            within the first 1MB */
                    0x10000,             /* Use 64KB stepping when searching */
                    NULL,
                    flashErase,
                    flashRead,
                    flashWrite,
                    NULL,                /* No reentrancy protection provided */
                    NULL,
                    NULL,
                    NULL,
                    FDC_DIAG_ALL);
      if (ret >= 0)
         break;

      /* Open failed. If not already formatted, then set format flag */
      /* and retry. */
      if (*formatted)
      {
         printf("FDC open failed\n");
         break;
      }
      format = 1;
   }

   return(ret);
}

U8 applGetAccess(void *userPointer)
{
   /* All access granted user */
   return(TFFS_ACCESS_SUPER);
}

int applTffsMount(...)
{
```

```
    U8 formatted;
    U8 format;
    TFFS_DISK_CTRL dapi;
    int fdcHandle;
    int ret;

    /* Initialize TFFS DISK API for using FDC */
    tffsFdcApiInit(&dapi);

    /* Assume that FDC does not have to be formatted */
    format = 0;
    for (;;)
    {
       /* Open FDC, do not force format */
       fdcHandle = applFdcOpen(format, &formatted);
       if (fdcHandle < 0)
       {
          ret = -1;
          break;
       }

       /* If FDC partition was formatted, then must TFFS format */
       if (formatted)
       {
          ret = tffsFormat(fdcHandle,
                           0,       /* No plans to expand in the future */
                           &dapi,   /* FDC API */
                           NULL,
                           NULL,    /* Not used in a preemptive environment */
                           NULL,    /* Not used in a preemptive environment */
                           TFFS_DIAG_ALL);
          if (ret < 0);
          {
             printf("TFFS format failed\n");
             break;
          }
       }

       /* Then mount TFFS disk */
       ret = tffsMount(fdcHandle,
                       &dapi,   /* FDC API */
                       NULL,
                       NULL,    /* Not used in a preemptive environment */
                       NULL,    /* Not used in a preemptive environment */
                       NULL,    /* Not used in a preemptive environment */
                       NULL,    /* Not used in a preemptive environment */
                       applGetAccess,
                       TFFS_DIAG_ALL);
       if (ret >= 0)
          break;

       /* If mount failed, and FDC partition has been formatted, then */
       /* there is something wrong.. . */
       if (formatted)
          break;

       /* Try to start from scratch... */
       format = 1;

       /* Must close opened FDC partition */
       fdcClose(fdcHandle);
    }

    return(ret);
 }
```

```
And the application would look something like below:

int tffsHandle;   /* Keep this for later use */

/* Must be done once before any use of TFFS/FDC */
fdcInit(1);
tffsInit(1);
```

```
tffsHandle = applTffsMount(...);
if (tffsHandle < 0)
{
   /* Can't use TFFS for some reason.... */
}
```