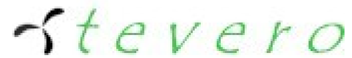


File Abstraction Layer (FAL) Description

Copyright Tevero AS
This document may be freely distributed as long as this copyright notice is not removed. The agreement for use is enclosed with the software source code described in this document.

Table of Contents

1 Introduction......3

1.1 Scope......3

1.2 Overview......3

1.3 Implemented File Functionality......3

2 FAL Principles......5

2.1 FAL Installation......5

2.2 FAL Usage......5

1 Introduction

1.1 Scope

The file abstraction layer is used on top of TFFS in order to

- provide a standard application programming interface (API) for file actions. Most C buffered file actions such as fopen(), fclose(), fread() etc. are supported. In addition some non-standard functions such as change directory, create directory etc. are provided.

- handle OS dependent issues. For instance, different threads/processes in a system may have different access privileges, and thus this information must be stored with the thread/process context.

- support multiple TFFS disk so that both RAM and flash disks can be included

- provide easy integration

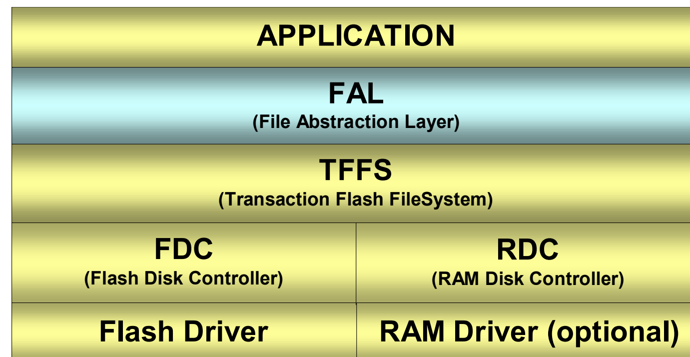
- gives full control over file access functionality, with the possibility to add special needs as required

Since TFFS, FDC and RDC are written OS independently, FAL must provide some glue functionality to make TFFS fit with the OS or application requirements. If no OS is used, or there is only one user, this gets real simple.

Due to the simplicity of FAL, the provided functions are not described in this document, but the use is described in the header for each function in the source files.

1.2 Overview

The below figure shows how FAL is used with TFFS and FDC/RDC in order to provide file access to a flash/RAM disk:



1.3 Implemented File Functionality

A brief overview of FAL implementation is shown below:

- Directories can be created/deleted/renamed (they must be empty to be deleted).
- A file can be created/deleted/renamed/copied.
- A file can be read at random locations.
- A file can be overwritten at random places (within existing file range).
- A file can have data appended to it.

- A file can be truncated.
- File/directory names can be max 28 characters.
- A file/directory can have read/write/hidden attributes (root dir has always READ only).
- Disks can be mounted on root directory “/” only. The root directory can only hold disks, no other files.
- Multiple TFFS disks supported.
- Optional support for ‘errno’ updating
- ASCII translation mode (CR -> CRLF) is not supported.

A directory structure could then typically look like this with a temporary RAM disk and a flash disk:

```
root -+-> flash -+-> sw -+-> appl.bin
      !                  !
      !                  +-> cfg.bin
      !
      +-> tmp -+--> snapshots -+-> 1.jpg

and one could access files using:

fopen("/tmp/snapshots/1.jpg", "rb");
fopen("/flash/sw/appl.bin", "rb");
```

2 FAL Principles

2.1 FAL Installation

FAL consists of multiple files, most of them rather small in order to provide optional inclusion in the target system if linked in from a library. (Most linkers will link in whole object files. A source file is normally compiled to an object file. So if only one function in a large source file with many functions is used, then all the unused functions may be included as well.)

Before use, *shrdconf.h/falconf.h* must be modified to suit the target in use. This is very simple, just do as the comments in *shrdconf.h/falconf.h* indicate. Notice that two functions to store and retrieve a pointer must be provided. If there is only one user in the system, then they would look like:

```
/* Ptr holding user state info for disk access */
static void *faltestStatePtr = NULL;

void *falTestGetStatePtr()
{
   return(faltestStatePtr);
}

void falTestSetStatePtr(void *ptr)
{
   faltestStatePtr = ptr;
}
```

For a multi-user (OS with threads/processes), the pointer must be stored in the thread/process context. Some OSs have a user defined area (or at least a pointer) per thread/process that can be used by the application. Another, although less optimal solution, is using a table search algorithm based on the thread/process ID etc.

If possible, retrieving a pointer should return NULL until it has been set explicitly. FAL protects against using NULL pointers.

In order to compile FAL, the TFFS include files *tffs.h* and *tffsconf.h* must be available in the include path.

2.2 FAL Usage

Prior to using FAL, the following must be done:

- disks and partitions must be opened and prepared for use (see TFFS/FDC/RDC documentation).
- TFFS disks opened must be mounted in FAL, using falMount() (unmount with falUnmount()).
- every user (thread/process) must register use of FAL using falRegister(). Since this action causes memory to be allocated, it is important to use falUnregister() before killing a thread/process.

The FAL API is not further documented here. Basic knowledge of standard C file handling is assumed. *fal.h* lists all the available functions, whereas the header of each function in the source files documents the usage.

FAL is made so that the user must only include the *fal.h* file, no other include file from FAL, TFFS, RDC or FDC are required for normal usage. (TFFS, RDC and FDC APIs are used for mounting and formatting disks.) This is also the reason why some FAL functions that could have been defined as macros were defined as functions instead.

Notice that *fal.h* automatically includes <stdio.h>, and then overrides some of the definitions. This can cause conflicts, depending on the environment, since there can be problems with the

redefinition done by FAL. (Should not be a problem with ISO/ANSI compliant compilers.) If such problems are encountered, they must be solved.

Another problem may be that the user wants to use certain functions both on the TFFS disk and on another IO device. For instance if the user wants to write both to a TFFS file and to stdout with fwrite():

```
/* Write to TFFS file and then the same to stdio */
fwrite("hi",  strlen("hi"), 1, fp);
fwrite("hi",  strlen("hi"), 1, stdout);
```

in the same source file. This can not be done, since FAL does not have any awareness of stdout and a fault will occur due to the "faulty" stdout stream. If such functionality is needed, this must be handled on a system to system base, either by expanding FAL functionality (catching the stdin, stderr and stdout devices in FAL), or by splitting source files.

At last, one should be aware of a subtler problem. One should not pass a stream pointer opened by FAL to a function that operates on non-FAL functionality. This can happen if one forgets to include the *fal.h* file in a source file, as illustrated below:

Source file *a.c:*

```
#include "fal.h"
extern void xyz(FILE *fp);

void abc()
{
   FILE *fp;
   /* This will use TFFS file functionality */
   fp = fopen("foo", "r");
   if (fp)
      xyz(fp);
}
```

Source file *b.c:*

```
#include <stdio.h>

void xyz(FILE *fp)
{
   /* Here we do not use TFFS file functionality!!! */
   while(!feof())
   {
      /* Read and process data */
      :
   }
}
```

The above will not give any compile time problems, but will be disastrous during execution.

Again, FAL was intended as a very simple wrapper. The FAL defines (fal.h) that redefine standard C functions can of course be removed and the actual falXXXXX() functions used directly in order to provide more unique names and avoid the above conflict.