SEcube™ FatFs library

Technical Documentation

Release: October 2021

Proprietary Notice

The present document offers information subject to the terms and conditions described hereinafter. The authors reserve the possibility to change the content and information described in this document and to update such information at any time, without notice. Despite the attention that has been taken in preparing this document, typographical errors, error or omissions may have occurred.

Authors

Matteo FORNERO *(Researcher, CINI Cybersecurity National Lab)* matteo.fornero@consorzio-cini.it
Nicoló MAUNERO *(PhD candidate, Politecnico di Torino)* nicolo.maunero@polito.it
Paolo PRINETTO *(Director, CINI Cybersecurity National Lab)* paolo.prinetto@polito.it
Gianluca ROASCIO *(PhD candidate, Politecnico di Torino)* gianluca.roascio@polito.it
Antonio VARRIALE *(Managing Director, Blu5 Labs Ltd)* av@blu5labs.eu

Trademarks

Disclaimer

Contents

1 Introduction

The **SE*cube*™** Open Source Security Platform[1] consists of a hardware device, the **SE*cube*™** , and of an open source SDK[2], including the firmware of the **SE*cube*™** and the libraries needed to implement host-side applications able to leverage the capabilities of the **SE*cube*™** itself.
From a hardware point of view, the **SE*cube*™** device embeds a micro SD card slot that can be used to expand the non volatile memory of the device. The firmware of the **SE*cube*™** , however, does not natively provide any straightforward interface to interact with the SD card. There is no support for any file system, because the SD card is considered as a very basic memory array.
In this context, handling files is very difficult. In order to solve the problem, the FatFs[3] library has been integrated inside the firmware of the **SE*cube*™** . FatFs is a library that has been developed to make embedded systems compatible with the FAT file system[4], one of the most simple file systems that are available. Thanks to the integration of FatFs, the firmware of the **SE*cube*™** is now able to interact with the SD card using an API that includes functions for managing standard files and encrypted files. This significantly expands the capabilities of the firmware of running more complex software, such as libraries that require non volatile configuration files (i.e., a database engine that is used by the firmware to store data in encrypted databases on the microSD card).
In general, the possibility of storing files (both standard and encrypted) is useful to overcome the physical limitations of the **SE*cube*™** and, specifically, the limited amount of internal flash memory.
The FatFs adaptation to the **SE*cube*™** that is presented in this document consists of a library that can be added to the standard **SE*cube*™** SDK; therefore, if you are not familiar with the SDK, please check out the official documentation[5].

2 FatFs

FatFs is a generic FAT/exFAT filesystem module for small embedded systems. The FatFs module is written in compliance with ANSI C (C89)[6] and completely separated from the disk I/O layer. FatFs is independent of the platform; therefore, it can be incorporated into small microcontrollers with limited resources, such as the **SE*cube*™** . Here is a recap of few interesting features of FatFs:

- DOS/Windows compatible FAT/exFAT filesystem
- Platform independent
- Very small footprint for program code and work area
- Long file name in ANSI/OEM or Unicode
- Multiple volumes (physical drives and partitions)
- File access functions
- Directory access functions
- File and directory management functions
- Volume management and system configuration functions

[1]https://www.secube.blu5group.com/
[2]https://github.com/SEcube-Project/SEcube-SDK
[3]http://elm-chan.org/fsw/ff/00index_e.html
[4]https://en.wikipedia.org/wiki/File_Allocation_Table
[5]https://github.com/SEcube-Project/SEcube-SDK
[6]https://en.wikipedia.org/wiki/ANSI_C

FatFs sits in the middle between the application and the storage device driver (i.e., the software that interacts with the storage device). There are four components that are required to integrate FatFs into an embedded device such as the **SE***cube*™ :

1. the FatFs module, meaning the library that must be integrated into the firmware of the embedded device;
2. the FatFs API, that is provided by the FatFs module to interact with the library itself;
3. the FatFs Media Access Interface (FatFs MAI), that is a set of functions whose prototypes are determined by the FatFs module and whose implementation must be done by the developer, since the MAI deals with the Storage Device Controls;
4. the Storage Device Controls, meaning the low-level software that interacts directly with the physical storage device; this layer is leveraged by the MAI to perform the operations requested by the FatFs module.

The stack of these four components is well illustrated in Figure 1 (image taken from the FatFs website[7]).

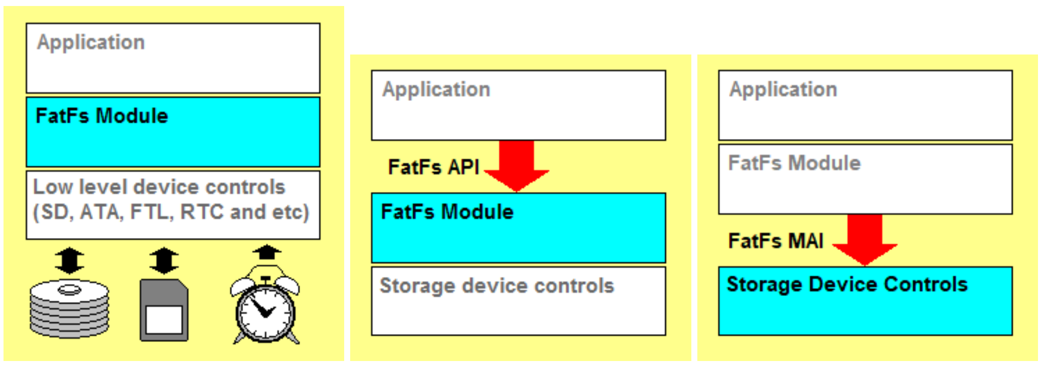

Figure 1: High level view of the FatFs stack.

3 General overview of the FatFs library for the SE*cube*™

Starting from the information presented in the previous section, FatFs has been integrated into the **SE***cube*™ firmware so that it sits in the middle between the high-level part of the firmware (i.e., the logic that handles the request coming from the computer to which the device is connected) and the low-level part of the firmware (i.e., the micro SD card driver). These layers are illustrated in Figure 2.
Notice that there is an additional component named 'SEFatFs', which is just some additional logic to FatFs so that the module can also handle encrypted files. The important aspect is that the application (i.e., a lightweight database engine running on the **SE***cube*™) leverages the API of FatFs (including the functions for managing encrypted files) to create, read, write, and delete files on the **SE***cube*™ micro SD.

[7]http://elm-chan.org/fsw/ff/00index_e.html

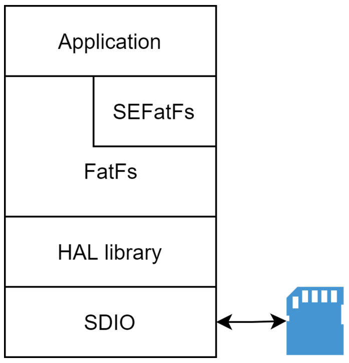

Figure 2: High level view of the FatFs stack.

4 Encrypted files

Since the **SE***cube*™ is a device focused on security, it would not make sense to integrate FatFs into the firmware without adding the capability of managing encrypted files. In this case, we are talking about files that are constantly encrypted on the micro SD, even while they are being read and written. This concept targets the security of 'Data-at-Rest', and it has been already embedded into another library for the **SE***cube*™ that is called **SE***file*™ [8]. While **SE***file*™ is a library that gives secure file system primitives to the applications running on the host side, the same secure file system primitives have been ported to the FatFs library.
Explaining how this is exactly implemented is out of the scope of this manual; however, the idea behind **SE***file*™ has been embedded into the FatFs API, with few tweaks and workarounds to make everything work correctly. For more information about the details of the **SE***file*™ library, please refer to the official documentation[9].

5 Limitations

- The current implementation does not support the management of directories.
- The current implementation does not support concurrent access to the micro SD card, this is a problem when the host system tries to access to the same file as the **SE***cube*™ .

6 Library setup

The FatFs library for the **SE***cube*™ has been developed and tested for the **SE***cube*™ SDK version 1.5.1 and 1.5.2. in order to add the library to the **SE***cube*™ firmware, you must follow these steps:

1. download the official **SE***cube*™ wiki[10];
2. follow the steps of the wiki to download and setup the **SE***cube*™ Open Source SDK (version 1.5.2 recommended);
3. once you are able to compile the code in the SDK and to run the examples, download the source code of the FatFs library[11];

[8] https://github.com/SEcube-Project/SEfile
[9] https://github.com/SEcube-Project/SEfile
[10] https://github.com/SEcube-Project/SEcube-SDK/blob/master/wiki/wiki_rel_012.pdf
[11] https://github.com/SEcube-Project/FatFs-for-SEcube

4. extract the content of the downloaded archive where you prefer;
5. copy the content of the extracted archive into the folder containing the firmware of the SDK that you compiled, paying attention to place the files in the right directories (i.e., place the content of the 'Inc' folder into the 'Inc' folder of the firmware, overwriting existing files);
6. compile the firmware;
7. flash the firmware on the **SE***cube*™ ;
8. start experimenting with the FatFs library.

7 FatFs for SE*cube*™ : API and data structures

Here we provide a simplified and high-level overview about the functions and data structures of this library. We are going to list only the functions needed to manage encrypted files, because all other functions are already provided by the FatFs library, so you can refer to the official documentation[12].
Notice that some of the functions listed below require flags and other particular parameters, which are the same as the standard functions of FatFs (i.e., the `secure_open()` function needs a combination of flags as third parameter, these flags are the same as the `f_open()` function of FatFs).

```
typedef struct
{
  FIL fp;
  uint32_t keyID;
  uint16_t algo;
  uint8_t decrypt_buffer[SE3_FATFS_LOGIC_DATA];
  uint16_t decrypt_buffer_size;
  uint8_t mode;
  bool dirty_bit;
  uint32_t pointer;
  uint8_t IV[SE3_FATFS_IV_LEN];
} SE3_FIL;
```

This structure is a wrapper around the standard `FIL` structure of FatFs. It is used to hold the information related to an encrypted file. While traditional FatFs functions work with a `FIL` structure, we need to add other data to manage the overhead of adding/removing encryption. It is not necessary to understand the meaning of each field, it is sufficient to know that this is the structure to be used with the functions listed in this section.

```
SE3_FRESULT secure_open(SE3_FIL* se_fp, char *path, BYTE mode,
   uint32_t keyID, uint16_t algo);
```

Open or create an encrypted file in the **SE***cube*™ micro SD; the encryption is completely managed by the library.

- The first parameter is a pointer to an empty SE3_FIL structure;
- the second parameter is the path of the file (plaintext, NULL terminated);

[12]http://elm-chan.org/fsw/ff/00index_e.html

- the third parameter is used to specify read only or read/write mode, as well as creation of a new file or opening of an existing file;
- the fourth parameter is the ID of the key to be used to encrypt the file
- the last parameter is the encryption algorithm to be used.

Notice that the last two parameters are ignored when opening an existing file (because the key ID and the algorithm have already been determined for that file).
In case of an encrypted file, the name of the file is also transformed into the SHA-256 digest of the original name.
It is not possible to specify the third parameter so that a file is created only if it does not exist and is opened if it exists; if you want to do that, you have to implement that mode manually (first checking if the file exists, and then passing the parameter accordingly).

```
SE3_FRESULT secure_close(SE3_FIL∗ se_fp);
```

Close a file that has been opened by `secure_open()`.

```
SE3_FRESULT secure_read(SE3_FIL∗ se_fp, uint8_t ∗dataOut,
   uint32_t dataOut_len, uint32_t ∗bytesRead);
```

Read the amount of bytes specified by the third parameter from the file specified as first parameter. Data is saved into the array pointed by the second parameter; the last parameter returns the amount of bytes that have been read. Notice that the caller is responsible for the allocation of the array specified as second parameter.

```
SE3_FRESULT secure_write(SE3_FIL∗ se_fp, uint8_t ∗dataIn,
   uint32_t dataIn_len);
```

Write the data pointed by the array specified as second parameter into the file passed as first parameter. The amount of data to be written is determined by the third parameter.

```
SE3_FRESULT secure_seek(SE3_FIL∗ se_fp, int64_t offset, uint32_t
    ∗position, uint8_t whence);
```

Move the position of the pointer inside the file specified as firt parameter of an amount of bytes equal to the second parameter, starting from the position specificed as fourth parameter. The final position inside the file is written into the integer specified as third parameter. Notice that, if a seek past the end of the file is requested, the pointer is set to the end of the file, instead of extending the size of the file.

8 Basic example

Here is a basic example of how to use the secure version of FatFs on the **SE***cube*™ firmware.

```
f_mount (&fs , "", 0); // mount the filesystem with FatFs
uint8_t buffOut [100];
SE3_FIL fp;
uint32_t bytes_read , seek_pos , input_len ;
char text [20] = " Hello world !";
input_len = strlen ( text );
```

```
secure_open (&fp , " example . txt", FA_CREATE_ALWAYS | FA_WRITE
    , 1, SE3_ALGO_AES_HMACSHA256 );
secure_write (&fp , ( void ∗) text , input_len );
secure_seek (&fp , 0, & seek_pos , SE3_FATFS_BEGIN );
secure_read (&fp , buffOut , input_len , & bytes_read );
secure_close (& fp);
```

9 FAQ

- **Can I access to the SE*cube*™ micro SD both from the SE*cube*™ and from the PC?** Yes, but race conditions could arise. FatFs is not designed to handle concurrent access in this scenario, so it is possible to get undefined behaviour and even to corrupt the FAT table. The easy solution is to block your PC from accessing to the **SE***cube*™ micro SD (this can be done in different ways, for example disabling the automount of the SD card), the complex solution is to implement some intermediate logic that handles concurrent access.