Preface

More information about JesFs is on my homepage.

I wrote JesFs for my very own needs and my daily work. JesFs was 100% designed for practical use in "**Small and Ultra-Low-Power IoT Devices**", that must be able to communicate over many different channels and must work reliable for years.

It is very convenient to talk with your devices over Internet. In any case much better than digging holes in the snow in 3000 mtr. altitude and -20°C (what for was the reason to start this project).

The JesFs File System is only the half way: it is only "really complete" with the JesFsBoot – a secure Bootloader - for (almost) all small CPU/MCUs, like ARM 32-Bit-cores, TI's MSP430/432, …

I'll add the docus for JesFsBoot later. Just as an info: JesFsBoot Is already available and ready for use, but I still need some time to finish the docus.

In any case: if you find JesFs (and soon JesFsBoot) useful, I'll be happy to support you. But: if you earn money with your system, I also would like to participate ;-) In return, you can count on my cooperation. As an Embedded System Engineer with several year of experience, I think I know how to play the game!

License Conditions

For non-commercial projects and tests, the use of my software is free of charge. However, as soon as you make profit with it, you need to license the use.

Currently (March 2018), my offer is: a license for JesFs (which also includes a license for JesFsBoot and all required software, like the JesFs Commander (for Windows), …) is 1 EUR per device (without VAT). I'll be happy to receive your mails ;-)

Technical Details – (Serial) NOR-Flash and JesFs

In NOR-Flash only "0" can be written. Normally NOR-Flash is organized in sectors (often something between 128 Bytes and 64k). Sectors can be erased in one Block only! Then all Bits turn to "1". Hence, an empty NOR-Flash has all Bytes set to "FF". Commonly for Serial NOR-Flashes the (smallest) sector size is 4kB (= 4096 Bytes).

Serial NOR-Flash normally is found in 8-pin ICs (like the MX25R6435F with 8*1 MBit (= 1 MB) as used on the CC131x-Launchpads from TI). Common Serial NOR Flash is currently (March 2018) available up to 16 MB.

Writing "0" is normally possible in any size, from single bits to larger blocks (often called "pages").

However: erasing a sector is possible only a limited counts (normally 1000-1000000). This makes the things a little bit trickier than on a “classical” Disk-Drive. The technical term for this is “wear leveling”: This means: try to erase a sector as seldom as possible (Reading is unlimited!).

JesFs was designed especially for NOR-Flash. Either as (external) serial chip or as (CPU internal) memory.

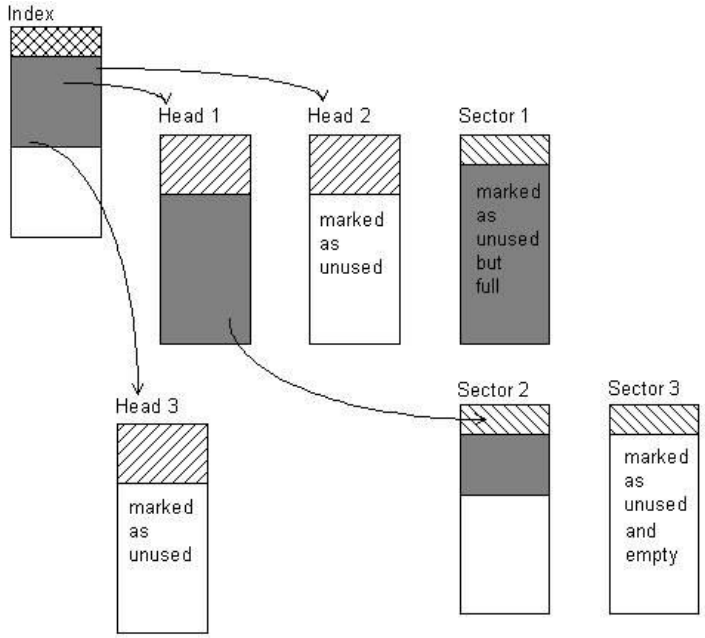
JesFs has 3 levels of usage for sector:

- Index (this is top level, only 1 can exist)
- Head (this is where a file starts)
- Sector (the “Pool”, for all the rest).

The Index is never erased! It holds the entries for the “Heads”. Each Head represents an either ACTIVE or DELETED file. For new files, JesFs always tries to recycle DELETED Heads. Only if necessary. Sectors from the “Pool” become Heads. Each Head has an entry in the Index. This means: the number of available files is limited by the size of the Index.

For an Implementation on Serial NOR-Flash with 4KB sectors of JesFs, the Index can hold up to ca. **1000** entries for Heads.

A Sector is only erased, if required.



Erasing a Sector takes several msec (see the data sheet of the chip). Hence, writing files is much slower than reading.

Serial NOR-Flash is known as very reliable (not like NAND-Flash, often used in USB-memories). Nevertheless JesFs is able to track a 32-Bit CRC for each file to ensure integrity! Using the the CRC32 costs only very little time for the CPU.

Another very important design topic of JesFs are:

Unclosed Files

This is something, that can not be found in traditional File Systems! Traditionally, every time a File is used (especially written), it must be closed, mainly to update the allocation tables and directories. This is something very sensitive: because if power is lost during “close”, the data of the file might be corrupt or -even worse- vanish into the “Data-Nirvana”…

JesFs has NO problems with unclosed files: Because empty NOR-Flash is always “FF”, the end of any file can always be found, as long as you don’t write “FF”s to it.

Hint: A traditional strategy to avoid “FF”s is, to use “Escape Bytes”: this means: If you want to write “FF”, simply write “FE 01” and for “FE” simply write “FE 00”. Or: write ASCII-Text… (Using “Escape Bytes” has an additional advantage, I’ll write something about this in a later document)

Conclusion: With JesFs: Close files that are “fixed”, and let the others open, keeping in mind, not to write “FF” and you will always be able to find your data.

Technical detail: JesFs can scan very quickly over Files to find the end (about 25 usec/Sector on a CC1310-Core). So even the end of a 16Mb File will be found in <100 msec!

JesFs Basics

JesFs is a “flat” File System. This means, there are no directories. The number of available files is only limited by either the size of the Index Sector or the total available number of sectors in the Flash Memory (this is e.g. 256 for the MX25R8035F with 1MB as used on the CC131x-Launchpad).

Each Filename can be up to 25 Characters long and can contain any character, except ‘\0’. So “$abc/hello\world&.bin$” would be a valid file name!

It is possible to have as many files open simultaneously as you want. The number of open files is only limited by the CPU’s memory, see below.. However: it is only allowed to write with one instance to each file (this is intuitive).

JesFs manages some Flags for the “Outer World”. Mainly these are “SF_OPEN_EXT_HIDDEN” and “SF_OPEN_EXT_SYNC”:

- SF_OPEN_EXT_HIDDEN means: This is a “private” File, that might hold passwords and security keys. The opposite is:
- SF_OPEN_EXT_SYNC: These files are marked to be synced to external Servers or others.

Details about External Access will follow in a separate docu.

JesFs consists of several Files:

- The Low Level Hardware driver: “JesFs_ll_xxxxx.c” (“JesFs_ll_tirtos.c” for the CC131x)
- The Hardware-Independent Mid- and High-Level driver: “jesfs_ml_hl.c”
- And the appropriate Header-Files. For the user, only “jesfs.h” is important.

Hint: In “jesfs.h” you may find all necessary data structures and error codes.

JesFS API

JesFS uses 2 types “Descriptors” for alle file access:

- File Descriptors: Each open File requires one. It holds all necessary data (like the current working Sector, current position, CRC, … Each File Descriptor requires about 28 Bytes (V1.0, see “JesFs.h”)

- Statistic Descriptors: to scan the Directory. Each Statistic Descriptor requires about 30 Bytes (V1.0, see “JesFs.h”)

A 3.rd variable (“sf_info”, size scalable from ca. 164-256 Bytes) holds all static info about the Serial NOR-Flash. That’s all!

Functions

```
int16_t fs_start(uint8_t mode);
```

Start (or Restart after Deepsleep) JesFS. Perform all necessary scans of the Flash and checks for errors. The more “FS_START_FASTL” is slightly faster than “FS_START_NORMAL”, see “JesFs.h”

```
void fs_deepsleep(void);
```

Put the Serial NOR-Flash to Deep Sleep, where it consumes only a few uA. Currently the MX25Rxxxx (as used on the CC131x-Launcpad) is top! Others take more…

```
int16_t fs_format(uint32_t f_id);
```

Dangerous: Erases the Serial NOR-Flash (which might take up to 2 minutes) and prepares JesFs. The parameter “f_id” is currently not used (in V1.0)

```
int32_t fs_read(FS_DESC *pdesc, uint8_t *pdest, uint32_t anz);
```

Reads Bytes from an open file to *pdest. If anz is larger than available, only the available bytes are read. It it possible to set pdest as NULL, then only nothing is read, but it can be used to finde the end of unclosed files or simply skip unwanted data. But ONLY for really read bytes (pdest <> NULL) the CRC of the file is updated.

```
int16_t fs_rewind(FS_DESC *pdesc);
```

Resets the file position to 0

```
int16_t fs_open(FS_DESC *pdesc, char* pname, uint8_t flags);
```

Opens a File (either for reading, deleting or writing in “unclosed mode” with the name pname and initialized the Descriptor. Possible Flags are “SF_OPEN_READ”, “SF_OPEN_RAW” or “SF_OPEN_CRC”.

```
int16_t fs_write(FS_DESC *pdesc, uint8_t *pdata, uint32_t len);
```

Opens a File for writing. If the file exists, old data is deleted. If the file does not exist, it will be created, if the Flasg “SF_OPEN_CRATE” is set. Optionally the Flag “SF_OPEN_CRC” and/or “SF_OPEN_EXT_HIDDEN” or “SF_OPEN_EXT_SYNC” can be used.

```
int16_t fs_close(FS_DESC *pdesc);
```

Only Files opened for Writing must be closed. By closing, the File Len and optionally the CRC is written.

```
int16_t fs_delete(FS_DESC *pdesc);
```

To delete a File, it must be opened in „SF_OPEN_RAW" mode. After deleting no fs_close() is required!

```
int16_t fs_info(FS_STAT *pstat, uint16_t fno );
```

This functions give access to the Directory. The Index contains a number of entries. For each entry (with number 'fno' the data can be retrieved. Hint: For deleted files still the name of the file is stored, but the data can not be accessed any more…

```
uint32_t fs_get_crc32(FS_DESC *pdesc);
```

Retrieves the CRC32 of the file (provided, that it is existent. Else, it simply will be "FFFFFFFF"). Can be used to make an error check of the file.

CRC32 (ISO 3309)

Checksums are crucial to ensure data integrity! The CRC32 is based on a proven industrial standard binary polynomial with an optimized "Hamming-Distance" to detect errors with maximum precision. A CRC32 ensures data integrity, but it is no encryption! You can use the functions also for your purposes.

The Demo

Very simple! The demo is based on a standard project on the CC13x0-Launchpad. Probably it will work with most others of the Wireless Simple-Link CPUs CC13xx/26xx Family too, but for my work, I used a CC1310- and a CC1350-Launchpad together with CCS Vers. 7.x.

Install the "TI-RTOS empty-Project" via Resource-Explorer and remove the File "empty.c". Instead, "Add Files", "Link" or "Copy" the Files from the "JesFs"-Demo", as shown on the following images.

After starting, you'll see a prompt '>' where you can test the API functions. Enjoy it!

Hint: The XDS110-Emulator of the CC1310-Launchpad offers a Serial COM-Port. However, I found it not very reliable. I prefer a separate COM-Port (based on a Low-Cost 3.3V-TTL-UART-cable form FTDI-CHIP, but many others ar OK too…).

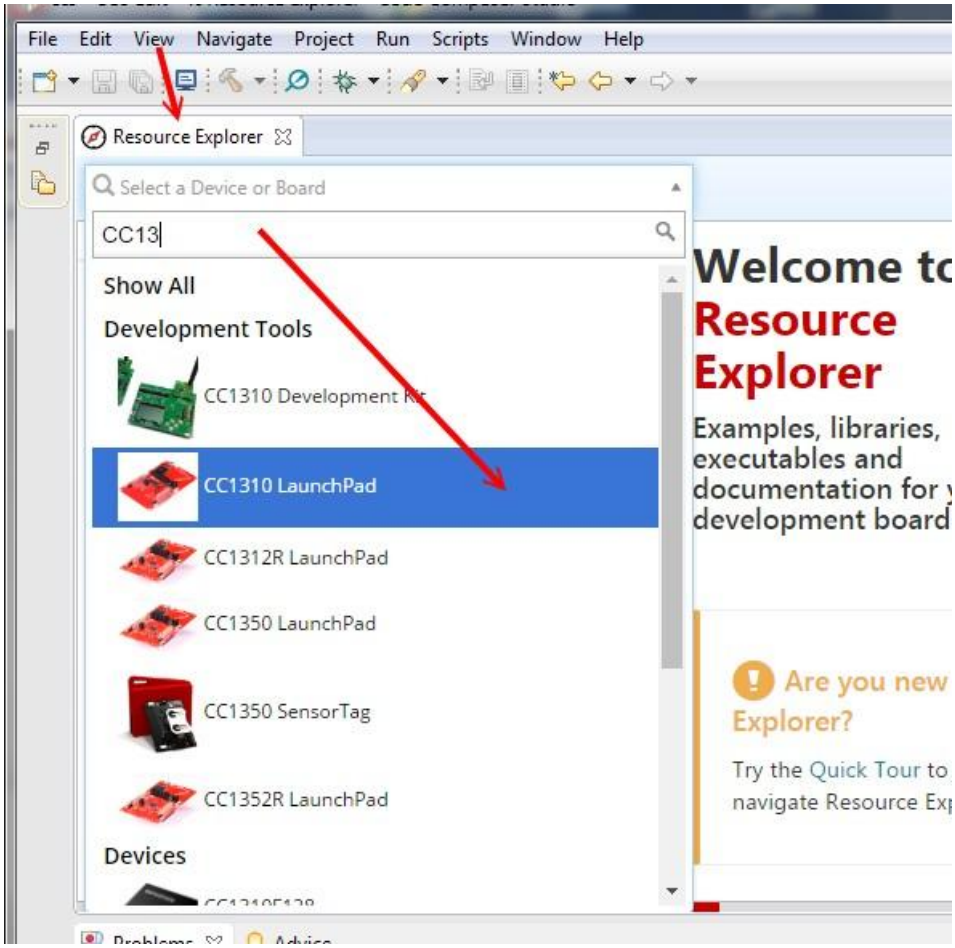
What comes next?

Very soon I will also publish the JesFsBoot – Secured Bootloader with AES128 encryption and direct interface to the Windows File System of your PC!

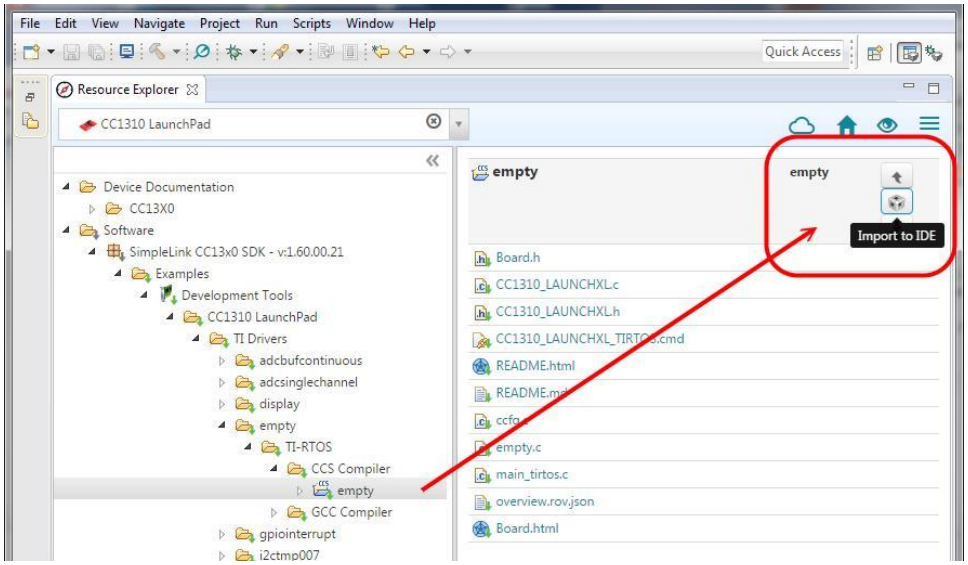
So: let's stay in touch!

Addendum: Demo images

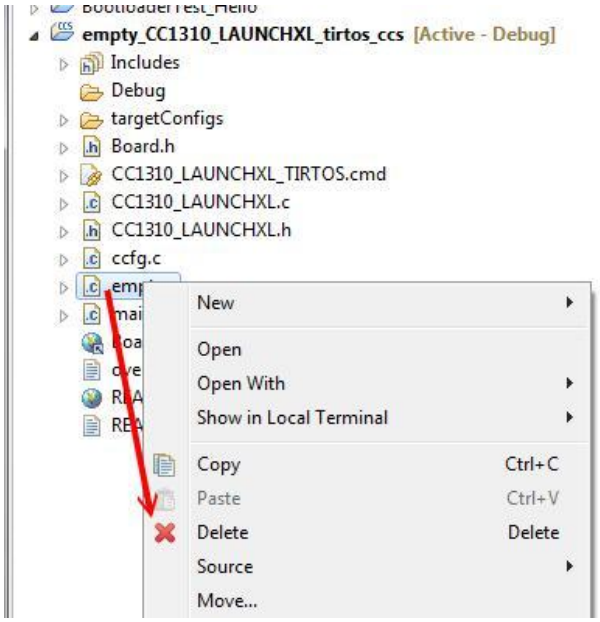
Running the Demo on an CC1310-Launcpad



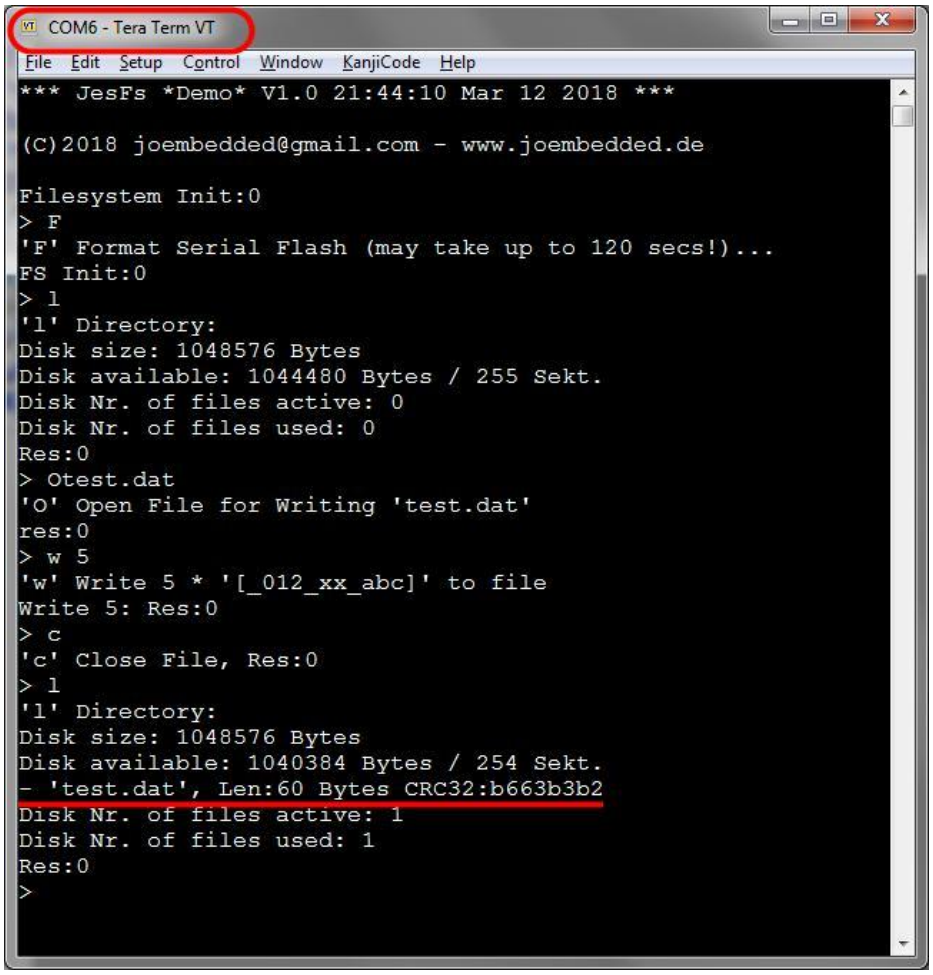
1.) Select your Launcpad in the Resource Explorer



2.) Import “empty”



3.) Delete “empty.c” and copy /add all JesFs-Files to the directory.



4.) Start a Terminal. Above you see the sequence:

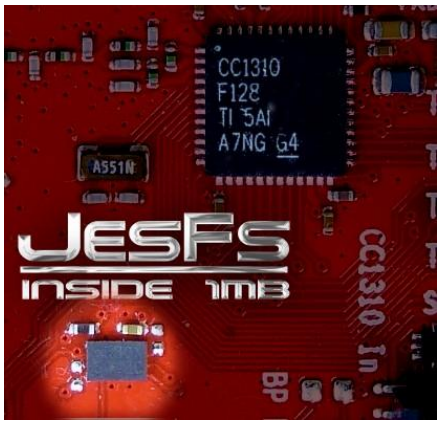
O test.dat opens “test.dat” for Writing

W 5 writes 5 times [_012_xx_abc]

c closes the file – and voila:

v (formerly ‘l’) shows it in the directory.

*** Have fun! And thanks for your Interest! ***
- Jo -



1MB – up to >200 Files – only 2x3 mm