EcFAT Getting started guide

Version 3.1.2

Table of contents

1 Introduction

The EmbCode FAT file system (EcFAT) is a file system driver intended for use in embedded systems. It provides the application programmer with an interface (API) which can be used to access individual files and directories on a storage device.

This document is intended for developers that are about to start using EcFAT. It describes how to compile EcFAT, how to select options and how to implement a suitable block driver. It also contains examples on how EcFAT is used.

Section 2 Overview describes how EcFAT works and how it relates to the rest of your application.

Section 3 Adding EcFAT to your application is a practical guide on how to add the EcFAT code to your application.

Section 4 Using EcFAT contains examples on how to use EcFAT.

2 Overview

2.1 Software layers in an application using EcFAT

In order to better understand how EcFAT fits in the rest of your application code the image below might be helpful.

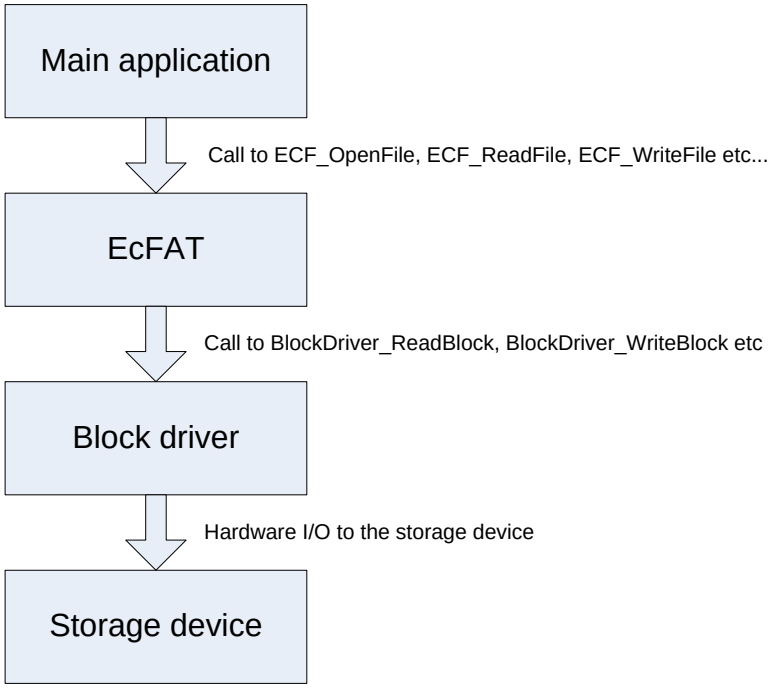


Figure 1

Main application:
This is the main application code. This is the part that needs to access files and that you are responsible for writing.

EcFAT:
The EmbCode FAT file system takes the high level file operations and translates them into simple block level reads and writes that the block driver can handle.

Block driver:
The block driver is called from EcFAT and is responsible for reading and writing data blocks to the actual hardware.

This is usually the EcFAT SD Card driver or a custom flash driver you've written for your specific flash chip.

Storage device:
This is the actual physical storage device that your data is stored on. It is usually an SD card or a flash memory chip.

2.1.1 Alternate configuration

EcFAT also supports accessing several file systems simultaneously which alters the model slightly. The most common case is that EcFAT connects to both a fixed storage device and a removable one. E.g. a flash chip soldered directly to the PCB and a removable SD card.

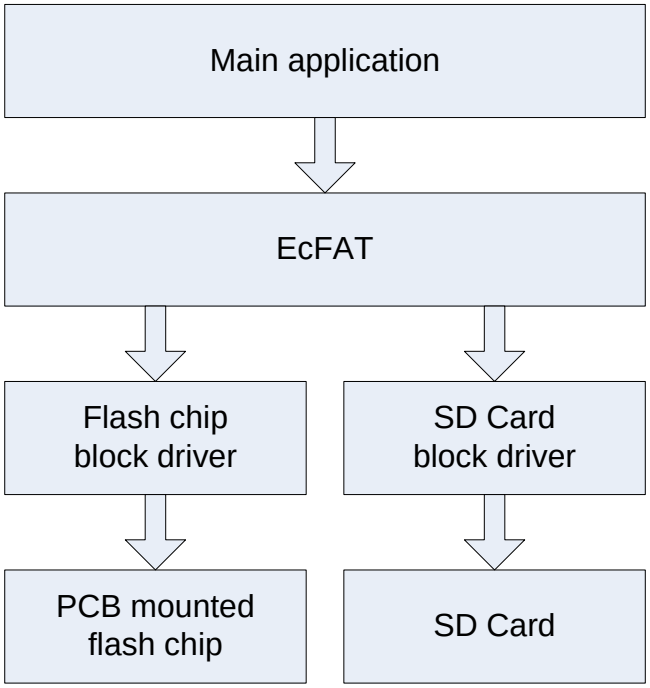


Figure 2

3 Adding EcFAT to your application

The files you receive as part of EcFAT comes packaged in a directory structure like this:

```
EcFAT 3.1.2/
      Documentation/
      EcFAT Explorer/
      Example EcFAT Options/
      Example Projects/
      Source/
```

To use EcFAT in your application you will need to do the following:

- Select the options for EcFAT
- Create a block driver or copy an existing one
- Include the EcFAT source files in your build environment in order to compile them.

3.1 Selecting options

EcFAT can be customized to fit your embedded system. Options are set in a file called *Project.h* which is included by EcFAT.

Project.h should be set in one of your configured include directories in order for EcFAT to find it.

3.1.1 Example options

Example options are available in the `Example EcFAT Options` folder. Example options are available for a *minimal* system, a *normal* system and a *large* system.

For a *minimal* system, all options that use extra RAM or ROM are disabled.
For a *normal* system, most functions are available but the size of the cache is limited.
For a *large* system, all functions are available and the cache is optimized for speed.

Below are the options for a *normal* system:

```
// Support long file names
#define ECF_OPT_SUPPORT_LONG_FILENAMES

// Support mounting of 2 drives simultaneously (A: and B:)
#define ECF_OPT_SUPPORTED_MOUNTPOINTS      2

// Support formatting of storage devices
#define ECF_OPT_SUPPORT_FORMAT

// Keep 2 sectors in the cache (will use 4*512 = 2048 bytes for cache)
#define ECF_OPT_SECTOR_CACHE               4
```

Individual options are explained in detail in the document *EcFAT API Reference*.

3.2 Setting up a block driver

A block driver is responsible for reading and writing the raw blocks of data from and to your storage device. EcFAT will call the block driver whenever it needs to read or write something.

Example block drivers are available in the `Source/Blockdriver` folder

3.2.1 How the block driver works

When EcFAT needs to access the storage device it will access the block driver through a struct called ECF_BlockDriver. This struct needs to be set up before the file system can be mounted.

You need to use at least four members of *struct ECF_BlockDriver*:

- The function pointer *m_fnReadSector* which points to a function that will read a sector from the storage device.
- The function pointer *m_fnWriteSector* which points to a function that will write a sector to the storage device.
- The function pointer *m_ fnGetVolumeInformation* which points to a function that will ECF will call to determine the sector size and total size of the storage device.
- The void pointer *m_BlockDriverData* which the block driver can use to store private data. ECF supplies it to the block driver when calling the three functions above.

Please note that the block driver usually have more functions than the three mentioned above. The other functions are used for e.g. initialization. These must of course be called appropriately before the block driver can be used by EcFAT.

See *EcFAT API Reference* for a more detailed explanation of the block driver functions.

3.2.2 Using the EcFAT SD card driver

If you plan to use the SD Card driver you will not need to write your own block driver. You just need to include SD Card driver source files in your project. By including these files you will have access to these functions:

```
ECF_ErrorCode SDCard_Init(void);

ECF_ErrorCode SDCard_ConnectToCard(void);

ECF_ErrorCode SDCard_ReadSector(
   struct ECF_BlockDriver *bd,
   DWORD sector,
   BYTE *data
)

ECF_ErrorCode SDCard_WriteSector(
   struct ECF_BlockDriver *bd,
   DWORD sector,
   BYTE *data
)

ECF_ErrorCode SDCard_GetVolumeInformation(
```

```
   struct ECF_BlockDriver *bd,
   WORD* pwSectorSize,
   DWORD* pdwNumberOfSectors
)
```

You will need to call SDCard_Init() and SDCard_ConnectToCard(void) before you attempt to mount a file system. A simple example might look like this:

```
   struct ECF_BlockDriver bd;

   ECF_Init();
   SDCard_Init();

   // Set up the blockdriver
   memset(&bd, 0, sizeof(struct ECF_BlockDriver));

   bd.fnReadSector           = SDCard_ReadSector;
   bd.fnWriteSector          = SDCard_WriteSector;
   bd.fnGetVolumeInformation = SDCard_GetVolumeInformation;

   ... make sure that an SD card is actually present ...

   if(SDCard_ConnectToCard() != SDCARD_STATUS_OK)
      halt("Can't connect to SD card");

   // Call ECF_Mount() to mount the file system.
   ...
```

3.2.3 Example RAM block driver

Below is the source code of a simple RAM block driver. It might not be really useful in a real system but it shows how to implement a block driver:

```
// Create a global to hold our data. Make it 32 kb
BYTE ramDriveData[64][512];

ECF_ErrorCode RAM_GetVolumeInformation(
   struct ECF_BlockDriver *bd,
   WORD* pwSectorSize,
   DWORD* pdwNumberOfSectors)
{
   *pwSectorSize = 512;
   *pdwNumberOfSectors = 64;

   return ECFERR_SUCCESS;
}

ECF_ErrorCode RAM_ReadSector(struct ECF_BlockDriver *bd, DWORD sector, BYTE
*data)
{
   if(sector >= 64)
      return ECFERR_PARAMETERERROR;
```

```
   memcpy(data, ramDriveData[sector], 512);

   return ECFERR_SUCCESS;
}

ECF_ErrorCode RAM_WriteSector(struct ECF_BlockDriver *bd, DWORD sector,
BYTE *data)
{
   if(sector >= 64)
      return ECFERR_PARAMETERERROR;

   memcpy(ramDriveData[sector], data, 512);

   return ECFERR_SUCCESS;
}
```

3.3 Including the EcFAT files in your project

If you are using EcFAT in a single project you can copy the entire `Source` folder to your project folder. If you're using EcFAT as a part of several different projects it is better include the `Source` folder in your include path.

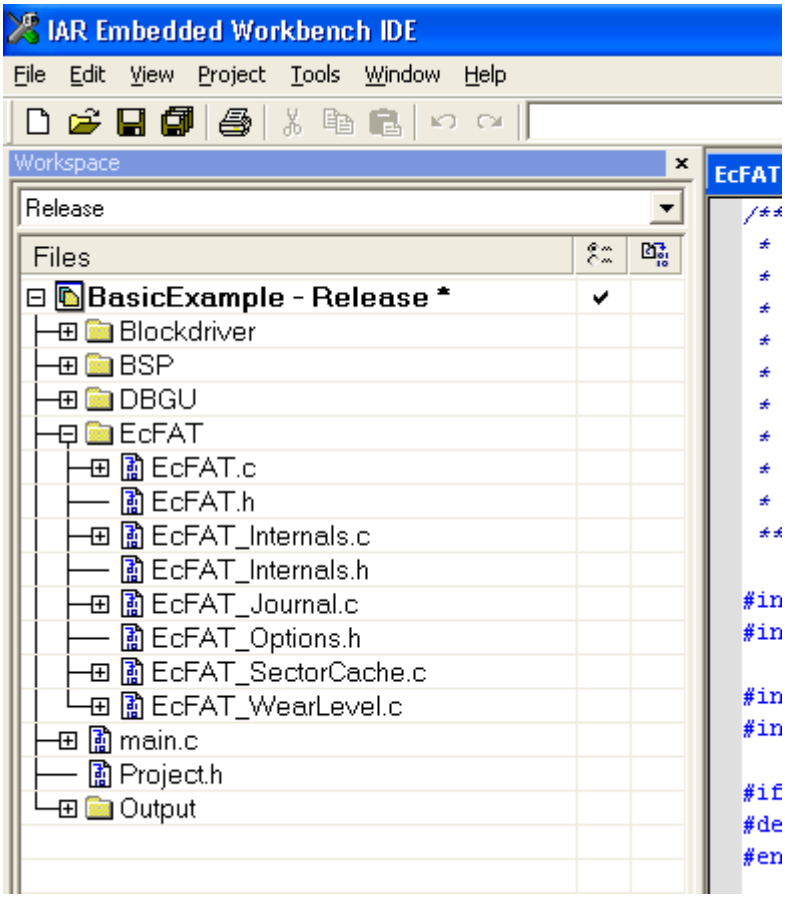
Either way, you can then start using EcFAT by adding:

```
#include <EcFAT/EcFAT.h>
```

to your source code. Also add the files:

EcFAT/EcFAT.c
EcFAT/EcFAT_Internals.c
EcFAT/EcFAT_Journal.c
EcFAT/EcFAT_SectorCache.c
EcFAT/EcFAT_WearLevel.c

as source files to your project. This will build and link EcFAT. Below is an example from IAR:



4 Using EcFAT

4.1 Initialization

In order to use EcFAT we must first initialise it to reset its internal state. Initialization is straightforward, we just call ECF_Init().

```
if(ECF_Init() != ECFERR_SUCCESS)
   halt("Can't initialize EcFAT");
```

4.2 Mounting a file system

To access a file system you will need to mount it. This will tell EcFAT which block driver to use and check that the storage device contains a valid FAT file system.

We'll start by filling out the ECF_BlockDriver struct which contains pointers to our block driver functions.

```
struct ECF_BlockDriver bd;

memset(&bd, 0, sizeof(struct ECF_BlockDriver));

bd.m_fnReadSector             = SDCard_ReadSector;
bd.m_fnWriteSector                      = SDCard_WriteSector;
bd.m_fnGetVolumeInformation   = SDCard_GetVolumeInformation;
```

This will provide information for EcFAT for the three basic functions it needs from the block driver.

We continue by calling ECF_Mount:

```
// Will mount partition 1 on the block device accessed by bd

if(ECF_Mount('A', &bd, ECF_MOUNT_PARTITION1) != ECFERR_SUCCESS)
   halt("Can't mount filesystem");
```

This will mount the first partition on the block device. This is the most common case when dealing with SD cards since they usually only contain one partition.

It will assign this mounted file system the drive letter 'A'.

In a matter very much like MS-DOS, EcFAT uses drive letters to distinguish between different drives. So to access a file called in "My file.txt" on the SD Card we just mounted we would refer to it as "A:\My file.txt".

If "My file.txt" was in the directory "My directory" we would access using the path "A:\My directory\My file.txt"

Remember that when entering the paths as C strings you need to escape \. So the path above needs to be entered as "A:\\My directory\\My file.txt".

4.3 Reading a file

In order to access a file we need to create an ECF_FileHandle. There is no need to initialize it, it will be cleared automatically when we open a file. Since we're going to read data, we'll also create a place to hold the data.

```
struct ECF_FileHandle fileHandle;
uint8_t data[32];
```

To open the file, we'll use our file handle, specify the file name and also the file mode.

```
if(ECF_OpenFile(&fileHandle, "A:\\My datafile.dat", ECF_MODE_READ)
     != ECFERR_SUCCESS)
   halt("Can't open file 'My datafile.dat'");
```

To read the first 32 bytes of the file we just call ECF_ReadFile.

```
if(ECF_ReadFile(&fileHandle, data, 32, NULL) != ECFERR_SUCCESS)
   halt("Can't read file");
```

Once we're done, we'll close the file handle.

```
if(ECF_CloseFile(&fileHandle)) != ECFERR_SUCCESS)
   halt("Can't close file");
```

Even if we only read data in this example, we need to check the return code of ECF_CloseFile(). The reason is that since EcFAT has a built in cache, ECF_Close() (or any other operation) might trigger a write operation for a previous read/write which we need to catch.

You may open and use as many files as you wish. You may also open the same file multiple times but you may only open one handle to a specific file in ECF_MODE_READ_WRITE or ECF_MODE_APPEND. The other file handles to that file must be in ECF_MODE_READ.

4.4 Writing a file

Writing data to a file is very similar to reading data. We'll just open the file in the ECF_MODE_READ_WRITE mode and call ECF_WriteFile() to actually write the data.

```
struct ECF_FileHandle fileHandle;

if(ECF_OpenFile(&fileHandle, "A:\\My datafile.dat", ECF_MODE_READ_WRITE)
     != ECFERR_SUCCESS)
   halt("Can't open file 'My datafile.dat'");

// Assume 'data' holds 32 bytes of data to write

if(ECF_WriteFile(&fileHandle, data, 32) != ECFERR_SUCCESS)
```

```
    halt("Can't write file");
```

It is possible to mix and match calls to ECF_ReadFile(), ECF_WriteFile() and ECF_SeekFile() on a file opened in the ECF_MODE_READ_WRITE mode. (ECF_SeekFile() changes the position of the file cursor)

When we are done, we close the file. EcFAT will automatically flush all the data to disk when a file is closed to make sure it is written.

```
if(ECF_CloseFile(&fileHandle) != ECFERR_SUCCESS)
   halt("Can't close file");
```

4.5 Scanning a directory

In order to list all the files and directories within another directory we'll need to create a struct ECF_FileHandle again. This is used by EcFAT to keep track of how much of the directory we've scanned so far. We'll also create a struct ECF_FileDirectoryData that EcFAT will fill with information about the files and directories we scan.

```
struct ECF_FileHandle         scanHandle;
struct ECF_FileDirectoryData  fileDirectoryData;
```

In order to start scanning a directory we'll call ECF_ScanDirBegin()

```
if(ECF_ScanDirBegin(&scanHandle, "B:\\My directory") != ECFERR_SUCCESS)
   halt("Can't scan path 'B:\\My directory'");
```

And then ECF_ScanDirNext() until we've scanned the entire directory:

```
while(
   ECF_ScanDirNext(&scanHandle, &fileDirectoryData) == ECFERR_SUCCESS
   )
{
   if(fileDirectoryData.m_dirAttr & ECF_ATTR_DIRECTORY)
      printf("Directory: %s\r\n", fileDirectoryData.m_szFileName);
   else
      printf("File: %s\r\n", fileDirectoryData.m_szFileName);
}
```

On each iteration ECF_ScanDirNext() will fill *fileDirectoryData* with all available information about the current directory entry.

If ECF_ScanDirNext is successful, it will return ECFERR_SUCCESS until there are no more entries and then return ECFERR_NOMOREFILES.

You may scan several directories at once. You must not, however, scan the same directory twice or from two threads as this will result in errors.

4.6 Flushing data

EcFAT will write data to the block driver whenever necessary. This is usually when writing and creating files but might also be when e.g. unmounting a file system.

EcFAT uses a cache in order to speed up the disk access. When creating or writing files, certain blocks that contain metadata about the files on the file system gets accessed often. By storing these blocks in a cache, speed is much improved.

Unfortunately, this also has a downside. When you call e.g. ECF_WriteFile() you can't be sure that the data you wrote has actually been written to the block driver, it might be sitting in the cache.

In order to force EcFAT to write (flush) all unwritten data it has in its cache to the block driver you can call ECF_Flush()

```
if(ECF_Flush('A') != ECFERR_SUCCESS)
   halt("Can't flush data to drive A:");
```

Once ECF_Flush() returns, all your data is guaranteed to be written to the block driver.

Use ECF_Flush() when you are expecting a power failure or when you just want to be sure that your data is written to the block driver.

But don't call ECF_Flush() too often since this will affect performance and wear the flash unnecessarily.

4.7 Unmounting

When you're done using a file system or are going to power down you should unmount your file system. This will flush all the changed data to the block driver.

```
if(ECF_Unmount('A') != ECFERR_SUCCESS)
   halt("Can't flush data to drive A:");
```

There is no need to un-initialize EcFAT. As long as you've unmounted all of your file systems, you can safely exit.

4.8 Formatting

If you are using an SD card it will probably already be formatted and you can just go ahead and mount it. But if you are using e.g. a flash chip that is soldered to your PCB you will need to format it before you can use it to store files.

We need to access the block driver directly to format it.

```
struct ECF_BlockDriver blockDriver;

// Initialize blockDriver, just like we did before ECF_Mount()
...
```

And then call ECF_Format() to do the actual formatting.

```
if(ECF_Format(&blockDriver, 2048, ECF_FORMAT_ALIGN|ECF_FORMAT_QUICK) !=
                ECFERR_SUCCESS)
   halt("Can't format drive");
```

This will format with a cluster size of 2048 bytes and align these to even 2048 byte boundaries. This example would be useful if your flash has a page size of 2048 bytes.

5 Journaling

5.1 What is it?

When you write files to a file system, the file system data structures often needs to be updated. This can be if you add a new file or write new data to an existing file. If the system crashes during this update or suffers from a power loss, these structure will become corrupted and the file system will become unusable.

Journaling protects the file system data structures and prevents them from being corrupted.

5.2 How does it work?

Journaling keeps a journal file that it uses when it needs to update the file system structures. When it is about to make an update to the file structure it first writes what is about to do into the journal. Once done, it writes the real changes to the file system. When the changes has been written to the file system, the entries in the journal are removed.

If say a power loss would occur during the time when the file structures happens, these will become corrupted. But this is where the journal comes in. The next time the file system is mounted, it will check the journal. If the journal contains an entry that has not been marked as completed, the write will be attempted again. This way, if a file system write is interrupted, it will automatically be completed the next time the file system is mounted.

EcFAT will create a hidden file called JOURNAL.ECF in the root folder which it will use to store the journal of the file system. This will ensure that the file system is still FAT compatible.

5.3 When to use journaling

You should use journaling in most situations to avoid data corruption.

You should not use journaling if:

- *Your system needs to be very quick and you are creating or are extending a lot of files.*

  The journaling process requires a few extra writes which will slow down the system somewhat.

- *When you do not want to create the JOURNAL.ECF file in the disc.*

  There can be special circumstances when you don't want this file on disc.

5.4 Implementing journaling

5.4.1 Define ECF_OPT_SUPPORT_JOURNAL

Example in Project.h:

```
#define ECF_OPT_SUPPORT_JOURNAL
```

5.4.2 Add the ECF_MOUNT_JOURNAL to your ECF_Mount() calls

Add the ECF_MOUNT_JOURNAL flag when mounting the file system. If the JOURNAL.ECF is not already on the file system, it will be automatically created.

Example:

```
err = ECF_Mount('A', &bd, ECF_MOUNT_PARTITION_AUTO|ECF_MOUNT_JOURNAL);
```

6 Wear-leveling

6.1 What is it?

Wear-leveling is a technique to even out the wear on flash memories. Flash memories are usually specified for a specific number of erase/writes cycles per sector so it is important to spread the writes across the entire area of the flash to make the flash last as long as possible.

File system like FAT were originally designed for floppy discs and hard drives. On these mediums, it doesn't matter how much you write a specific sector, they are not more likely to fail because of it. Because of this, when writing data to a FAT file system, there tends to be a lot of writes concentrated in a few sectors and the flash wears out in these sectors.

6.2 When to use wear leveling

You should use wear leveling when:

- You are writing data to a flash that doesn't have built-in wear-leveling. This is typically serial or parallel flash circuit that you solder on to a PCB.

You should NOT use wear leveling when:

- Do not use it with SD cards and USB memory sticks.

  These already have built-in wear-leveling. Also note that a regular PC will not be able to read a wear-leveled file system so only use it on devices that are not accessible for the end user.

- Do not use it if your system frequently restart/remount your file system.

  EcFAT keeps track of statistics to remember how often a sector has been written. This normally stays in RAM. But if you frequently restart/remount your file system you force this statistics to be written to disc and the advantage of the wear-leveling is lost.

- Do not use it if your system doesn't have enough resources.

  Keeping track of the write count and moving blocks around takes more time. Even if you write just a few hundred bytes, this might trigger a sector to be relocated which in total will cause around 3 writes.

Keep in mind that wear leveling will also use part of your disc for metadata. How much depends on the disc size but generally you will lose around 4-5% of disc space.

6.3 What can you expect from wear-leveling?

It depends very much on the pattern of writes from your application. But in a typical case you can expect the top number of writes to be about 100 times less.

One way of testing this is to use the ECF_GetHighestWriteCountSeen() function.

Let's say you plan for your system to work for 10 years. Run your system under "normal" conditions under 1 week (and restart it if you expect it to be reset once a week). Call

ECF_GetHighestWriteCountSeen() and record the value. If the value is say 90 then you can extrapolate this to 90 * 10 years * 52 weeks/year = 46 800 writes. So you will need a flash that will allow at least 46 800 writes per sector.

You can also contact EmbCode and describe your file write pattern and we can give you an estimate.

6.4 Implementing wear leveling

6.4.1 Define ECF_OPT_SUPPORT_WEARLEVEL

Example in Project.h:

```
#define ECF_OPT_SUPPORT_WEARLEVEL
```

6.4.2 Set up the size of the wear-level meta block cache (optional)

Define ECF_OPT_WEARLEVEL_META_CACHE and set it to the number of meta blocks you want in your cache. Each meta block will take around 530 bytes of RAM regardless of the chosen sector size.

For each 256 physical sectors, 1 meta block will be used. Not all of them needs to stay in RAM but excessive reads/writes to the meta block will remove the benefit of wear-leveling. Below are the recommendations based on the number of sectors:

128-258 physical sectors:
Cache of 1 meta block. There is no need for more.
259-514 physical sectors:
Cache of 2 meta blocks. There is no need for more.
515-770 physical sectors:
Cache of 3 meta blocks. There is no need for more.
771-1026 physical sectors:
Cache of 4 meta blocks. There is no need for more.
1027-1282 physical sectors:
Cache of 4 meta blocks. There is no need for more but up to 5 is useful.
1283-1538 physical sectors:
Cache of 4 meta blocks. There is no need for more but up to 6 is useful.
1539-1794 physical sectors:
Cache of 4 meta blocks. There is no need for more but up to 7 is useful.
1795 physical sectors or more:
Cache of 4 meta blocks. There is no need for more but up to 8 is useful.

Example in Project.h:

```
#define ECF_OPT_WEARLEVEL_META_CACHE 8
```

If you choose not to define ECF_OPT_SUPPORT_WEARLEVEL, it will automatically be set to 4.

6.4.3 Add support in your blockdriver for ECF_*SECTOR_512_BYTES_ONLY flags

This is optional if you are using only 512 byte sectors.

EcFAT will only use 512 bytes of your sectors when reading/writing metadata. If you use sectors that are 1024-4096 bytes, your block driver needs to respect the flags. The flags are needed to save RAM in the metadata cache.

If ECF_BlockDriver::m_fnReadSector is called with ECF_READSECTOR_512_BYTES_ONLY you should only write the first 512 bytes of the sector into the buffer.

Note that the buffer is only 512 bytes so if you write an entire sector to the buffer you will corrupt the memory after the buffer which will likely cause a crash.

If ECF_BlockDriver::m_fnWriteSector is called with ECF_WRITESECTOR_512_BYTES_ONLY you should only write the first 512 bytes of the buffer to the sector on the disk. You may leave the remaining bytes at any value.

Note that if you read past the 512 bytes, you risk a bus fault and writing random data to disk.

6.4.4 Wear-level format your block device to enable wear-leveling

Call ECF_WearLevelFormat() on the block devices you want to use. This should only be called once and will automatically wear-level the entire block device (even if it has several partitions).

It will reset the wear-leveling statistics so it should only be called once per device.

6.4.5 Continue the initialize the block device as normal

Continue to use and initialize the block device as normal (e.g. using ECF_CreatePartition(), ECF_Format(), ECF_Mount() etc). ECF will automatically detect that it is wear leveled.

6.4.6 Check usage statistics (optional)

If you wish to get statistics during development or get a warning for a worn flash, periodically call ECF_GetHighestWriteCountSeen() to get an estimate for how many writes the mostly written sector has.

7 Bad block management

7.1 What is it?

In order to increase the yield of flash memories, they are sometimes shipped with some blocks being faulty on delivery. A typical flash memory might ship with up to 80 bad blocks.

This needs to be handled by the file system.

7.2 How does it work?

A file system or a block device driver typically reserved some space for blocks to use as replacement for the blocks that are or go bad. It also stores a table with information on which blocks have been remapped. This is quite complicated due to the fact that blocks may also be bad among the replacement blocks and in the remapping table.

7.3 When to use bad block management

Use bad block management whenever you have a device that is shipped with bad blocks. You may also use it when blocks are likely to go bad during the usage of the device. EcFAT supports remapping blocks during normal operation so blocks are allowed to go bad during the actual usage of the device.

7.4 Implementing bad block management

7.4.1 Enable wear-leveling

Bad block management is implemented in the wear-leveling layer so you need to start by enabling wear-leveling.

7.4.2 Define ECF_OPT_SUPPORT_BAD_BLOCK_MANAGEMENT

Example in Project.h:

```
#define ECF_OPT_SUPPORT_BAD_BLOCK_MANAGEMENT
```

7.4.3 Set up how many remaps EcFAT can hold in RAM (optional)

Define `ECF_OPT_WEARLEVEL_MAX_BAD_BLOCK_COUNT` to specify how many bad blocks relocations EcFAT can hold in RAM. This value needs to be higher or equal to the maximium bad block count of any disk you wear-level format of mount with this driver.

Example in Project.h:

```
#define ECF_OPT_WEARLEVEL_MAX_BAD_BLOCK_COUNT 512
```

ECF_OPT_WEARLEVEL_MAX_BAD_BLOCK_COUNT will default to 256 if not specified.

7.4.4 Wear-level format your block device to enable wear-leveling

Specify the maximum number of blocks that can go bad when calling ECF_WearLevelFormat().

8 Trim support

8.1 What is it?

EcFAT supports Trim which means that EcFAT will let the underlying block driver know if a sector is used or if it can be erased.

The normal use case of Trim is to pre-erase the flash so that a write can happen immediately without waiting for an erase. But it can also be used if the underlying driver benefits from knowing which sectors are unused and can be thrown away.

8.2 Implementing trim support

To add Trim support to a block driver you need to supply the function m_fnTrimSectorRange() to the blockdriver. You should also check and act on the ECF_WRITESECTOR_IS_TRIMMED flag passed to m_ fnWriteSector().

m_fnTrimSectorRange() will be called when EcFAT wants to free a single or a range of sectors.

If m_fnWriteSector() is called with the ECF_WRITESECTOR_IS_TRIMMED flag, EcFAT is writing a sector that should be trimmed. EcFAT will keep track of which sectors are trimmed but since power losses and program crashes happen, it is still best to check that the sector is actually erased before writing to it.

See the documentation for m_fnTrimSectorRange() and m_fnWriteSector() in the API Reference for more information and example code.

9 Support for larger flash blocks

9.1 What is it?

EcFAT supports two typical scenarios where the sector size differs from the smallest erasable block size in a flash memory.

9.1.1 Using 512 bytes FAT sectors on a flash with a 4KB block sizes

A lot of flash chips (and especially the cheaper ones) has a minimum erasable block size of 4KB.

This is usually not ideal for small systems for several reasons:

- Each cache entry takes 4KB which leads small system to only use 1 or 2 sectors of cache which is often inefficient.
- Reading small chunks of data is slower since 4KB always has to be read from disk even if only a few hundered bytes are needed.
- Disk usage can become quite high if you have a lot of smaller files since the smallest allocatable amount of flash is 4KB.

9.1.2 Flash block sizes larger than 4KB

Since the FAT file system does not support larger sector sizes than 4KB, it was previously unrealistic to use EcFAT with them.

It was possible to write a driver that “faked” smaller sectors that the flash block size by using a RAM buffer. But this breaks journaling since it is no longer true that a single write can only destroy the sector being written to. It also break wear-levelling since the wear-levelling layer will write different sectors that actually translate to the same flash block which will get the count wrong.

9.2 How does it work?

The block driver reports to EcFAT how many sectors are used for each flash block. EcFAT then uses this information to write block in an order that the block driver can write directly to the flash in without using a RAM buffer.

There are a lot of circumstances when EcFAT needs modify a sector that is part of a larger block where EcFAT needs to rewrite several sectors around the modified block. In order to do this and to not lose these blocks during a power loss, journaling must be enabled in order for EcFAT to be able to use the journal as temporary storage during such a rewrite.

9.3 When to use

The mode is very useful for smaller systems where there isn’t much RAM. It is also useful with flash memories where the block size is larger than 4KB.

The down-side is that it is a lot slower than then normal mode. EcFAT needs to be very careful to write a whole block at a time and write all the sectors of it in order. A lot of the times these writes must therefore occur through the journal (even for data writes) which is very slow compared to a normal write.

The disk must be formatted with EcFAT 3.1 or later for this mode to work. Otherwise the sectors and the journal will not be aligned to the flash block size. It is also recommended that you set the cluster size to the size of your flash block if possible.

9.4 Implementing

To use a different sector size than the native flash block size you need to modify the block driver to report the desired sector size and the number of sectors per flash block.

For a flash memory with 4KB native sectors but where you wish to use 512 byte sectors the block driver should report a sector size of 512. The number of sectors should be reported to be the number of 512 byte sectors available.

Two examples:

Scenario	Flash size	Flash block size	Number of flash blocks	Sector size reported by block driver	Number of sectors reported by block driver	Sectors per block reported by block driver
Flash has 4KB sectors but you want to use 512 byte sectors	4 MByte	4 Kbyte	1024	512	4MB/512 = 8192	4KB/512 = 8
Flash has 256KB blocks but you want 4KB sectors	256 MByte	256 KByte	1024	4 096	256MB/4096 = 65536	256 KB/4 KB = 64

You must also implement m_fnGetDriveProperty in struct ECF_BlockDriver and return the ECF_DRIVEPROPERTY_SECTORSPERBLOCK property. For a flash with 4KB native sectors and 512 byte virtual sectors, the number of sectors per block is 4096 / 512 = 8.

When m_fnWriteSectors is called the block driver must also erase a flash block whenever an aligned write is made. For all sectors that are not aligned, the block driver can assume that the sector is already erased.

So a write of sector 8 would first trigger an erase of the flash block that starts with sector 8 (sector 8 to 15) and then write the 512 bytes needed for sector 8. After this point, if EcFAT wants to write this flash block again, EcFAT may either write sector 9 (to continue writing the flash block) or write sector 8 again to trigger a new erase and write of sector 8.

This implies that a write to sector 8 will erase sector 9..15. This is the intended effect. When EcFAT issues such a write, it will make sure that there is a backup of sector 9..15 elsewhere on the disk.

You also need to enable journaling and mount the drive with the ECF_MOUNT_JOURNAL flag.

You may or may not use wear-leveling depending if you need it.