ECF Getting started guide

Version 2.2.1

Table of contents

1 Introduction

The EmbCode FAT file system (ECF) 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 ECF. It describes how to compile ECF, how to select options and how to implement a suitable block driver. It also contains examples on how ECF is used.

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

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

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

2 Overview

2.1 Software layers in an application using ECF

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

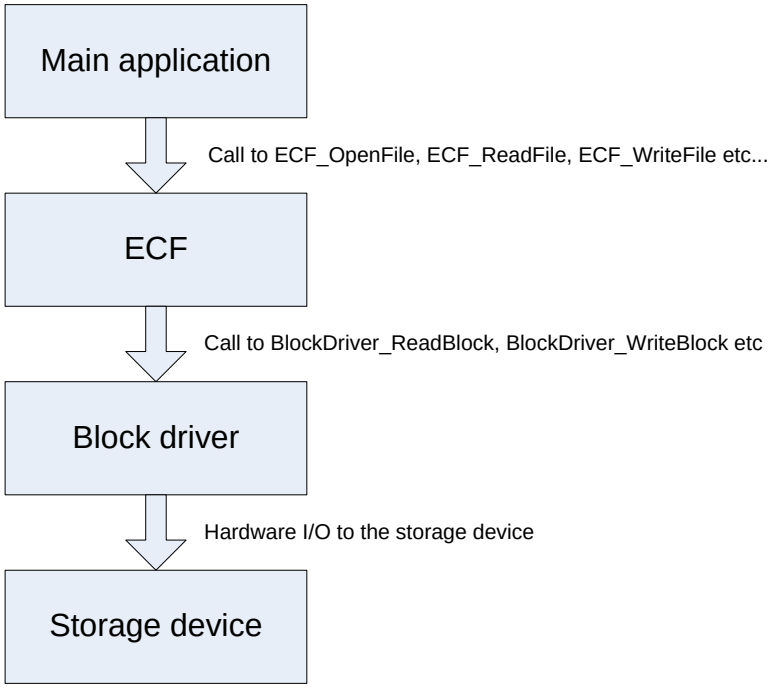


Figure 1

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

ECF:
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 ECF and is responsible for reading and writing data blocks to the actual hardware.

This is usually the ECF 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

ECF also supports accessing several file systems simultaneously which alters the model slightly. The most common case is that ECF 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.

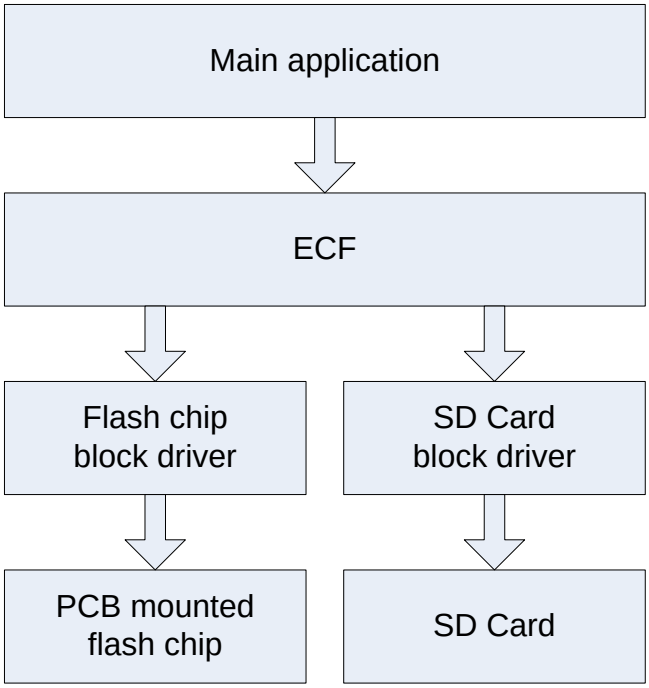


Figure 2

3 Adding ECF to your application

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

```
ECF 2.2.1/
      Documentation/
      ECF Explorer/
      Example ECF Options/
      Example Projects/
      Source/
```

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

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

3.1 Selecting options

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

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

3.1.1 Example options

Example options are available in the `Example ECF 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 *ECF 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. ECF 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 ECF 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 ECF.

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

3.2.2 Using the ECF 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 ECF files in your project

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

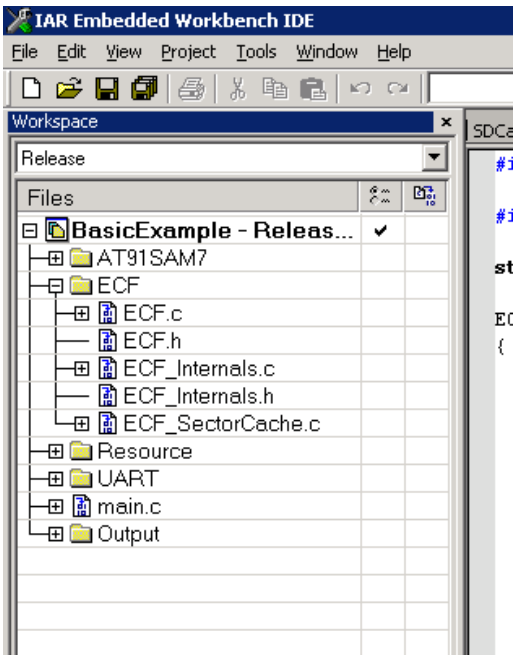
Either way, you can then start using ECF by adding:

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

to your source code. Also add the files:

ECF/ECF.c
ECF/ECF_Internals.c
ECF/ECF_SectorCache.c

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



4 Using ECF

4.1 Initialization

In order to use ECF 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 ECF");
```

4.2 Mounting a file system

To access a file system you will need to mount it. This will tell ECF 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 our block driver functions.

```
struct ECF_BlockDriver bd;

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

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

This will provide information for ECF 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, ECF 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 string 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;
BYTE 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) != 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 ECF 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

To write 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 possibly 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. ECF 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 ECF to keep track of how much of the directory we've scanned so far. We'll also create a struct ECF_FileDirectoryData that ECF 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

ECF 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.

ECF 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 ECF 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 ECF. As long as you've unmounted all your file system, 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 bd;

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

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

```
   if(ECF_Format(&bd, 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 Implementing Trim support

ECF supports Trim which means that ECF will let the underlying 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.

To add Trim support to a block driver you need to supply the function fnTrimSectorRange() to the blockdriver. You may also supply fnWriteTrimmedSector().

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

If fnWriteTrimmedSector() is supplied, it will be called instead of fnWriteSector() when ECF writes a sector that should be trimmed. ECF 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.

Also see the documentation for fnTrimSectorRange() and fnWriteTrimmedSector() in the API Reference.