ECF API Reference

Version 2.2.1

Table of contents

1 General operations

1.1 ECF_Init

The ECF_Init function initialises the ECF file system driver. It must be called before any of the other functions can be called.

```
ECF_ErrorCode ECF_Init(void);
```

Parameters

None

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

If you haven't defined ECF_OPT_USE_MUTEX and aren't using a multithreaded system, ECF_Init() cannot fail and there is no need to check the return code.

Remarks

There is no need to uninitialise ECF. Just make sure you have unmounted all the drives when you exit.

Example Code

See ECF_Mount

1.2 ECF_GetErrorMessage

The ECF_GetErrorMessage function translates an ECF_ErrorCode to a readable string.

```
const char * ECF_GetErrorMessage(
   ECF_ErrorCode err
);
```

Parameters

err
This is the ECF_ErrorCode to translate.

Return value

Returns a const string that can be displayed to the end user.

Remarks

Error codes between ECFERR_BLOCKDRIVER_ERROR_FIRST and ECFERR_BLOCKDRIVER_ERROR_LAST are reserved for block driver errors and ECF_GetErrorMessage will not return a meaningful error message for these error codes.

Example Code

See ECF_Mount

2 Block device operations

2.1 ECF_Mount

The ECF_Mount function mounts a file system.

```
ECF_ErrorCode ECF_Mount(
   char cDriveLetter,
   struct ECF_BlockDriver *pBlockDeviceDriver,
   WORD wFlags
);
```

Parameters

cDriveLetter
This is the drive letter you want to use to refer to this file system. E.g. 'A'

pBlockDeviceDriver
This is a pointer to the ECF_BlockDriver struct that allows the file system to access your block device.

wFlags
These are flags specifying which partition to mount.

`ECF_MOUNT_PARTITION_AUTO:`
Attempts to mount partition 1 if a partition table exists but will mount partitionless if not. This is the default and recommended for most applications.

`ECF_MOUNT_PARTITION1:`
Mounts partition 1 on the block device. This is usually the case for mounting an SD Card.

`ECF_MOUNT_PARTITION2:`
Mounts partition 2 on the block device.

`ECF_MOUNT_PARTITION3:`
Mounts partition 3 on the block device.

`ECF_MOUNT_PARTITION4:`
Mounts partition 4 on the block device.

`ECF_MOUNT_PARTITIONLESS:`
Mounts a block device that does not contain a partition table. This is usually the case when mounting a block device that resides on an embedded flash.

Return value
Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks
This must be called before accessing files on the disk.

Example Code

```
#include <stdio.h>
```

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

// Use this buffer as a 32kb RAM Disk
BYTE abRamDisk[64][512];

ECF_ErrorCode RamDriver_ReadSector(
   struct ECF_BlockDriver *,
   DWORD dwSector,
   BYTE *pbData)
{
   memcpy(pbData, abRamDisk[dwSector], 512);
   return ECFERR_SUCCESS;
}

ECF_ErrorCode RamDriver_WriteSector(
   struct ECF_BlockDriver *,
   DWORD dwSector,
   BYTE *pbData)
{
   memcpy(abRamDisk[dwSector], pbData, 512);
   return ECFERR_SUCCESS;
}

ECF_ErrorCode RamDriver_GetVolumeInformation(
   struct ECF_BlockDriver *,
   WORD*  pwSectorSize,
   DWORD* pdwNumberOfSectors)
{
   *pwSectorSize = 512;
   *pdwNumberOfSectors = 64;
   return ECFERR_SUCCESS;
}

int main(int argc, char **argv)
{
   struct ECF_BlockDriver bd;
   ECF_ErrorCode err;

   ECF_Init();

   memset(&bd, 0, sizeof(bd));
   bd.fnReadSector = RamDriver_ReadSector;
   bd.fnWriteSector = RamDriver_WriteSector;
   bd.fnGetVolumeInformation = RamDriver_GetVolumeInformation;

   err = ECF_Format(&bd, ECF_FORMAT_QUICK);
   if(err != ECFERR_SUCCESS) {
      printf("Block device could not be formatted. Error: %s\r\n",
        ECF_GetErrorMessage(err));
      return 1;
   }

   err = ECF_Mount('A', &bd, ECF_MOUNT_PARTITION_AUTO);
   if(err != ECFERR_SUCCESS) {
      printf("Block device could not be mounted. Error: %s\r\n",
   ECF_GetErrorMessage(err));
      return 1;
   }
```

```
   // ... Read or write some files to the disk ...
   err = ECF_Unmount('A');
   if(err != ECFERR_SUCCESS) {
      printf("Block device could not be unmounted. Error: %s\r\n",
   ECF_GetErrorMessage(err));
      return 1;
   }
}
```

See also

ECF_Format, ECF_Unmount

2.2 ECF_Unmount

The ECF_Unmount function unmounts a file system. It is very important to unmount a file system after usage so that all the data is saved.

```
ECF_ErrorCode ECF_Unmount(
   char cDriveLetter
);
```

Parameters

cDriveLetter

This is the drive letter of the file system you wish to unmount. E.g. 'A'.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

This must be called when you are done using a drive. If you are worried about power loss or similar scenarios you do not need to call ECF_Unmount()/ECF_Mount() repeatedly. Call ECF_Flush() instead to write all data to disk.

Example Code

See ECF_Mount

See also

ECF_Mount, ECF_Flush

2.3 ECF_Format

The ECF_Format function formats a block device to prepare it to hold files. It will erase all existing data on the block device.

```
ECF_ErrorCode ECF_Format(
   struct ECF_BlockDriver *pBlockDeviceDriver,
   WORD wClusterSize,
   WORD wFlags
);
```

Parameters

pBlockDeviceDriver
This is a pointer to the ECF_BlockDriver struct that allows the file system to access your block device.

wClusterSize
This is the desired cluster size. Valid values are 512, 1024, 2048, 4096, 8192, 16384 and 32768.

wFlags
Options to ECF_Format(). Several options can be used and are OR:ed together.

Specify only one or none of the ECF_FORMAT_PARTITIONLESS, ECF_FORMAT_CREATE_PARTITION1 and ECF_FORMAT_PARTITIONx flags. If none of these flags is specified, ECF_FORMAT_PARTITIONLESS will be used as the default.

`ECF_FORMAT_PARTITIONLESS:`
Format this block device without using a partition table. Recommended setting when formatting an internal flash and you only want to use one partition. This is the default.

`ECF_FORMAT_CREATE_PARTITION1:`
This will clear the partition table, create a partition that occupies the entire block device and format it. Recommended setting when formatting an SD card and you only want to use one partition.

`ECF_FORMAT_PARTITION1:`
This will format partition 1. The partition must already exist.

`ECF_FORMAT_PARTITION2:`
This will format partition 2. The partition must already exist.

`ECF_FORMAT_PARTITION3:`
This will format partition 3. The partition must already exist.

`ECF_FORMAT_PARTITION4:`
This will format partition 4. The partition must already exist.

`ECF_FORMAT_QUICK:`
Performs a quick format by not clearing the data area of the disk when formatting.

Note: If you are using Trim support, the entire area will always be trimmed regardless of this flag.

`ECF_FORMAT_ALIGN:`

Aligns the cluster placement to the cluster size. This is useful if you are using flash memory to store your file system. By using this flag and a suitable cluster size you can be sure that each of the clusters is aligned to an even page boundary on your flash.

As an example, if you are using a flash with a page size of 4096 bytes it is recommended that you enable the ECF_FORMAT_ALIGN flag and set the cluster size to 4096 for best results.

`ECF_FORMAT_ONLY_FAT12:`

Will force the FAT12 format. This will possibly waste space and create a FAT12 that is as big as possible. It is useful if you want to make sure the formatted disk is compatible with ECF Lite.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

ECF_Format will erase all the data on the block device. It needs to be called for an unformatted block device before it can be mounted.

You need to specify ECF_OPT_SUPPORT_FORMAT in your Project.h for ECF to compile with support for this function.

Example Code

See ECF_Mount

See also

ECF_Mount, ECF_CreatePartitionTable, ECF_CreatePartition

2.4 ECF_CreatePartitionTable

The ECF_CreatePartitionTable function creates an empty partition table. If one exists, it will be overwritten.

```
ECF_ErrorCode ECF_CreatePartitionTable(
   struct ECF_BlockDriver *blockDevice,
   WORD wFlags
);
```

Parameters

blockDevice
This is a pointer to a struct ECF_BlockDriver of the disk you want to create the partition table on.

wFlags
Flags. No flags are currently defined, specify 0.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

You need to specify ECF_OPT_SUPPORT_FORMAT in your Project.h for ECF to compile with support for this function.

Example Code

```
void InitializeDisk(struct ECF_BlockDriver *blockDriver)
{
   // Error checking omitted, sector size of 512 assumed.

   // Initialize the partition table
   ECF_CreatePartitionTable(blockDriver, 0);

   // Create partition 1: A 2 MiB FAT partition for configuration
   ECF_CreatePartition(blockDriver, ECF_PARTITION_TYPE_FAT,
2*1024*1024/512, 0);

   // Create partition 2: A 10 MiB FAT partition for logs
   ECF_CreatePartition(blockDriver, ECF_PARTITION_TYPE_FAT,
10*1024*1024/512, 0);

   // Create partition 3: The rest of the space as a RAW partition
   // that we write data to directly
   ECF_CreatePartition(blockDriver, ECF_PARTITION_TYPE_RAW, 0, 0);

   // Format partition 1
   ECF_Format(blockDriver, 512, ECF_FORMAT_PARTITION1);

   // Format partition 2
   ECF_Format(blockDriver, 512, ECF_FORMAT_PARTITION2);
}
```

See also

ECF_CreatePartition, ECF_Format

2.5 ECF_CreatePartition

The ECF_CreatePartition function creates a partition on the supplied block device.

```
ECF_ErrorCode ECF_CreatePartition(
   struct ECF_BlockDriver *blockDevice,
   BYTE pbPartitionType,
   DWORD dwSizeInSectors,
   WORD wFlags
);
```

Parameters

blockDevice
This is a pointer to a struct ECF_BlockDriver of the disk you want to create the partition on.

pbPartitionType
The partition type:

`ECF_PARTITION_TYPE_FAT:`
Create a FAT partition to store a FAT file system on.

`ECF_PARTITION_TYPE_RAW:`
Create a RAW partition to store raw data in.

dwSizeInSectors
The size of the partition in sectors. Specify 0 to use all of the remaining space.

wFlags
Flags. No flags are currently defined, just specifiy 0.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

Call ECF_CreatePartitionTable() first to create/clear the partition table. Then call ECF_CreatePartition() for each partition you want to create.

You need to specify ECF_OPT_SUPPORT_FORMAT in your Project.h for ECF to compile with support for this function.

Example Code

See example for ECF_CreatePartitionTable

See also

ECF_CreatePartitionTable, ECF_Format

2.6 ECF_GetPartitionInfo

The ECF_GetPartitionInfo function returns information about a partition on a block device.

```
ECF_ErrorCode ECF_GetPartitionInfo(
   struct ECF_BlockDriver *blockDevice,
   BYTE bPartitionNo,
   BYTE *pbPartitionType,
   DWORD *pdwStartSector,
   DWORD *pdwPartitionSizeSectors
);
```

Parameters

blockDevice
This is a pointer to a struct ECF_BlockDriver of the disk for which you want the partition information.

bPartitionNo
The number of the partition you wish to get info for. 1-4 are valid values.

pbPartitionType
A pointer to a BYTE that will receive the partition type

pdwStartSector
A pointer to a DWORD that will receive the start sector of the partition.

pdwPartitionSizeSectors
A pointer to a DWORD that will receive the size of the partition in sectors.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Returns ECFERR_NOPARTITION if the specified partition does not exist.

Returns ECFERR_NOPARTITIONTABLE if a partition table does not exist.

Remarks

None.

3 File system operations

3.1 ECF_Flush

ECF_Flush writes all unsaved data in the sector cache to the block device.

```
ECF_ErrorCode ECF_Flush(
   char cDriveLetter
);
```

Parameters

cDriveLetter
This is the drive letter you want to use to refer to this file system. E.g. 'A'.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

Some systems that can experience sudden power loss can benefit from calling ECF_Flush when it wants to make sure that all data has been written to the block device.

After a successful call to ECF_Flush the data is guaranteed to be written to the block device.

Example Code

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

void WriteToLogFile(struct ECF_FileHandle *fileHandle, const char
*szLogEntry)
{
   // Error checking omitted

   ECF_WriteFile(&fileHandle, strlen(szLogEntry), szLogEntry);

   // Make sure the log entry is actually written to disk.
   // We assume that the file is located on drive 'A'.
   ECF_Flush('A');
}
```

See also

ECF_Mount, ECF_Flush

3.2 ECF_CalculateFreeSpace

The ECF_CalculateFreeSpace function calculates the amount of free disk space on a mounted filesystem.

```
ECF_ErrorCode ECF_CalculateFreeSpace(
   char cDriveLetter,
   DWORD *pdwFreeSectors,
   WORD *pwSectorSize
);
```

Parameters

cDriveLetter
This is the drive letter of the file system you which to calculate the free space of. E.g. 'A'

pdwFreeSectors
This is a pointer to a DWORD that will receive the number of free sectors.

pwSectorSize
This is a pointer to a WORD that will receive the sector size. This parameter is not mandatory and may be NULL.

Return value
Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks
ECF_CalculateFreeSpace needs to scan the entire FAT table to get an accurate count of the number of free sectors which may take some time.

4 File operations

4.1 ECF_OpenFile

ECF_OpenFile opens a file on a mounted file system.

```
ECF_ErrorCode ECF_OpenFile(
   struct ECF_FileHandle *pFileHandle,
   const char *filename,
   BYTE bMode
);
```

Parameters

pFileHandle

This is a pointer to a struct ECF_FileHandle structure to hold data about the open file. The contents of the struct ECF_FileHandle will be cleared. There is no need to initialize it.

filename

This is the name of the file to open. Files are always specified with their full paths including the drive letter. To open a file called "Log.txt" on drive A you will need to specifiy the filename: A:\Log.txt (which is "A:\\Log.txt" when entered as a C string)

To open a file called "My file.data" in the directory "My folder" on drive B you need to specify the filename: B:\My folder\My file.data ("B:\\My folder\\My file.data" as a C string)

The maximum total path length including the trailing NUL character is 260 characters.

bMode

Specifies in which mode the file should be opened. Supported modes are:

`ECF_MODE_READ:`

Opens the file for reading. Reading will start at the beginning of the file.

`ECF_MODE_READ_WRITE:`

Opens the file for reading and writing. Reading and writing will start from the beginning of the file. If the file doesn't exist it will be created.

`ECF_MODE_APPEND:`

Opens the file for reading and writing. Writing the file will write to the end of it. If the file doesn't exist it will be created.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

If the call to ECF_OpenFile was successful you can now use your file handle to call other file operations.

You may open the same file several times but you may only open it once in write mode. This enables you to have one process that logs data to a file while another process reads it.

Example Code

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

void main(int argc, char **argv)
{
   // ...Mount filesystem on 'A' here...

   struct ECF_FileHandle fileHandle;
   const char *cszMessage = "This will be written to the file\n";

   if(ECF_OpenFile(&fileHandle, "A:\\Log.txt", ECF_MODE_APPEND)
      == ECFERR_SUCCESS)
   {
      // Error checking omitted
      ECF_WriteFile(&fileHandle, cszMessage, strlen(cszMessage));
      ECF_CloseFile(&fileHandle);
   }
}
```

See also

ECF_ReadFile, ECF_WriteFile, ECF_SeekFile, ECF_GetFileSize, ECF_CloseFile

4.2 ECF_CloseFile

ECF_CloseFile closes an open file handle.

```
ECF_ErrorCode ECF_CloseFile(
   struct ECF_FileHandle *pFileHandle
);
```

Parameters

pFileHandle
The pointer to an open file handle.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

Note: If you have opened a file in read/write mode, ECF will often write to the block device when a file is closed. Since this might fail, it is important to check the error code even when you close the file.

Example Code

See example for ECF_OpenFile()

See also

ECF_OpenFile

4.3 ECF_ReadFile

ECF_ReadFile reads data from an open file.

```
ECF_ErrorCode ECF_ReadFile(
   struct ECF_FileHandle *pFileHandle,
   BYTE *data,
   UINT size
);
```

Parameters

pFileHandle
A pointer to an open file handle.

data
A pointer to a buffer big enough to hold the read data.

size
The number of bytes to read.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

None.

Example Code

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

#define COPYFILE_BUFFERSIZE   4096

void CopyFile(const char *cszSource, const char *cszDestination)
{
   struct ECF_FileHandle srcFile;
   struct ECF_FileHandle destFile;
   DWORD dwDataLeftToCopy;
   DWORD dwDataToCopyNow;
   BYTE abCopyBuffer[COPYFILE_BUFFERSIZE];

   // Error checking omitted
   ECF_OpenFile(&srcFile, cszSource, ECF_MODE_READ);
   ECF_OpenFile(&destFile, cszDestination, ECF_MODE_READ_WRITE);

   ECF_GetFileSize(&srcFile, &dwDataLeftToCopy);

   while(dwDataLeftToCopy)
   {
      dwDataToCopyNow = COPYFILE_BUFFERSIZE;
      if(dwDataToCopyNow > dwDataLeftToCopy)
         dwDataToCopyNow = dwDataLeftToCopy;

      ECF_ReadFile(&srcFile, abCopyBuffer, dwDataToCopyNow);
      ECF_WriteFile(&destFile, abCopyBuffer, dwDataToCopyNow);
```

```
      dwDataLeftToCopy -= dwDataToCopyNow;
   }
   ECF_CloseFile(&destFile);
   ECF_CloseFile(&srcFile);
}
```

See also

ECF_OpenFile

4.4 ECF_WriteFile

ECF_WriteFile writes data to an open file.

```
ECF_ErrorCode ECF_WriteFile(
   struct ECF_FileHandle *pFileHandle,
   BYTE *data,
   UINT size
);
```

Parameters

pFileHandle
A pointer to an open file handle.

data
A pointer to a buffer that holds the data to be written.

size
The number of bytes to write.

Return value
Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks
If you attempt to write past the end of the file, the file will be extended to hold all the written data.

Example Code
See example for ECF_ReadFile.

See also
ECF_OpenFile

4.5 ECF_SeekFile

ECF_SeekFile changes the position of the file cursor within a file.

```
ECF_ErrorCode ECF_SeekFile(
   struct ECF_FileHandle *pFileHandle,
   DWORD dwPosition
);
```

Parameters

pFileHandle
A pointer to an open file handle.

dwPosition
The absolute position within the file to move the file cursor to.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

This function only moves the cursor to an absolute position within the file. If you want to move it relative to the current position or relative to the end of the file, you need to call ECF_GetFileSize() and ECF_GetFilePosition() to calculate where to move.

See also

ECF_GetFilePosition, ECF_GetFileSize

4.6 ECF_GetFileSize

ECF_GetFileSize function gets the size of a currently open file.

```
ECF_ErrorCode ECF_GetFileSize(
   struct ECF_FileHandle *pFileHandle,
   DWORD *pdwFileSize
);
```

Parameters

pFileHandle
A pointer to an open file handle.

pdwFileSize
A pointer to a DWORD to hold the file size.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

This can only be used to retrieve the size of an open file. To get the size of a file on disk, use ECF_GetFileInfo instead.

See also

ECF_GetFileInfo

4.7 ECF_GetFilePosition

ECF_GetFilePosition function retrieves the position of the cursor in a currently open file.

```
ECF_ErrorCode ECF_GetFilePosition(
   struct ECF_FileHandle *pFileHandle,
   DWORD *pdwFilePosition
);
```

Parameters

pFileHandle
A pointer to an open file handle.

pdwFilePosition
A pointer to a DWORD to hold the file position.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

None.

4.8 ECF_Rename

The ECF_Rename function renames a file or directory.

```
ECF_ErrorCode ECF_Rename(
   const char *oldPath,
   const char *newPath
);
```

Parameters

oldPath
This is the path of the file or directory to rename.

newPath
This is the new path of the file or directory.

The maximum total path length including the trailing NUL character is 260 characters.

Return value
Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks
oldPath and newPath must be on the same drive.

The subdirectories in the new path does not need to exist. If they don't exist, they will be created.

You must not rename an open file or a directory that is being scanned.

4.9 ECF_Delete

The ECF_Delete function deletes a file or directory.

```
ECF_ErrorCode ECF_Delete(
   const char *path
);
```

Parameters

path
This is the path of the file or directory to delete.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

If a directory is given as argument, ECF_Delete() will also recursively remove all the directories and files contained within that directory.

4.10 ECF_PathExists

The ECF_PathExists function checks if a path exists on a mounted filesystem.

```
ECF_ErrorCode ECF_PathExists(
   const char *path,
   BOOL *bDirectory
);
```

Parameters

path
The path to check.

bDirectory
A pointer to a BOOL which will be set to TRUE if the supplied path is a directory. This parameter is not mandatory and may be NULL.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success).

ECFERR_PATHNOTFOUND will be returned if the path does not exist.

Remarks

None.

4.11 ECF_GetFileInfo

The ECF_GetFileInfo function retrieves information about a specific file or directory.

```
ECF_ErrorCode ECF_GetFileInfo(
   const char *path,
   struct ECF_FileDirectoryData *fileDirectoryData
);
```

Parameters

path
This is the path to the file or directory to retrieve data about.

fileDirectoryData
This is a pointer to a struct that will be filled with information about the file or directory.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

None.

5 Directory operations

5.1 ECF_CreateDirectory

The ECF_CreateDirectory function creates a new directory.

```
ECF_ErrorCode ECF_CreateDirectory(
   const char *path
);
```

Parameters

path
This is the name of directory to create. E.g. "A:\My new directory" ("A:\\My new directory" as a C string)

The maximum total path length including the trailing NUL character is 260 characters.

Return value
Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks
None.

5.2 ECF_ScanDirBegin

The ECF_ScanDirBegin function starts the scan of a directory.

```
ECF_ErrorCode ECF_ScanDirBegin(
   struct ECF_FileHandle *scanDirPosition,
   const char *path
);
```

Parameters

scanDirPosition

This a pointer to a struct ECF_FileHandle that ECF will use to keep track of the directory scan. You do not need to initialize the struct, ECF_ScanDirBegin will initialize it for you.

path

This is the path to scan. E.g. "A:\My directory" ("A:\\My directory" as a C string).

The maximum total path length including the trailing NUL character is 260 characters.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Remarks

None.

Example Code

```
void ListDirectory(const char *path)
{
   ECF_ErrorCode err;
   struct ECF_FileHandle scanHandle;
   struct ECF_FileDirectoryData fileData;

   // Error checking omitted
   ECF_ScanDirBegin(&scanHandle, path));

   while(ECFERR_SUCCESS == ECF_ScanDirNext(&scanHandle, &fileData))
   {
      // Skip the entry if it starts with .
      if(fileData.m_szFileName[0] == '.')
         continue;

      if(fileData.m_dirAttr & ECF_ATTR_DIRECTORY)
         printf("%s <DIR>\r\n", fileData.m_szFileName);
      else
         printf("%s\r\n", fileData.m_szFileName);
   }
}
```

See also

ECF_ScanDirNext

5.3 ECF_ScanDirNext

The ECF_ScanDirNext function retrieves information about the next file or directory in a directory scan.

```
ECF_ErrorCode ECF_ScanDirNext(
   struct ECF_FileHandle *scanDirPosition,
   struct ECF_FileDirectoryData *fileDirectoryData
);
```

Parameters

scanDirPosition
This is a pointer to the struct previously initialized by ECF_ScanDirBegin.

fileDirectoryData
This struct will be filled with information about the next available file/directory.

Return value

Returns one of the ECF error codes (ECFERR_SUCCESS on success)

Returns ECF_NOMOREFILES when all the entries in the directory have been scanned.

Remarks

When you scan a directory, the special entries "." and ".." (for the current and the parent directory) will be returned. Most users want to ignore these so be sure to check for these names.

Example Code

See example for ECF_ScanDirBegin().

See also

ECF_ScanDirBegin

6 Data structures

6.1 struct ECF_BlockDriver

struct ECF_BlockDriver is used by ECF_Mount() and ECF_Format() to get access to the storage device.

```
struct ECF_BlockDriver
{
     ECF_ErrorCode (*fnReadSector)(struct ECF_BlockDriver *,
             DWORD sector, BYTE *data);
     ECF_ErrorCode (*fnWriteSector)(struct ECF_BlockDriver *,
             DWORD sector, BYTE *data);
     ECF_ErrorCode (*fnGetVolumeInformation)(struct ECF_BlockDriver *,
            WORD* pwSectorSize, DWORD* pdwNumberOfSectors);

     ECF_ErrorCode (*fnTrimSectorRange)(struct ECF_BlockDriver *,
             DWORD dwStartSector, DWORD dwEndSector);
     ECF_ErrorCode (*fnWriteTrimmedSector)(struct ECF_BlockDriver *,
             DWORD sector, BYTE *data);

     void *m_BlockDriverData;
};
```

Members

fnReadSector:
This is a pointer to a function that ECF can call to read a sector from the storage device.

fnWriteSector:
This is a pointer to a function that ECF can call to write a sector to the storage device.

fnGetVolumeInformation:
This is a pointer to a function that ECF can call to get information about the sector size and total size of the storage device.

fnTrimSector:
This is a pointer to a function that ECF can call to trim a sector. This is optional and can be NULL.

fnWriteTrimmedSector:
This is a pointer to a function that ECF can call to write a trimmed sector. If provided, this function will be called when ECF writes a sector that has previously been trimmed. This is optional and can be NULL even if m_fnTrimSectorRange is not NULL. If not provided, m_fnWriteSector will be called instead.

m_BlockDriverData:
A void pointer that can be used by the block driver to store private data. ECF will pass a pointer to the ECF_BlockDriver struct when it calls any of the functions above. The block driver can use these to access its private data.

Remarks

These functions must be created by the user. An instance of the struct ECF_BlockDriver must be cleared, filled with pointers to these functions and passed to ECF_Mount() and ECF_Format().

If your block driver only supports one instance, you don’t need to use m_BlockDriverData, you can just use global variables instead.

6.1.1 fnReadSector

fnReadSector reads a single sector from the block device (usually an SD card or flash memory).

```
ECF_ErrorCode fnReadSector(
   struct ECF_BlockDriver *bd,
   DWORD sector,
   BYTE *data
);
```

Parameters

bd

This is a pointer to the struct ECF_BlockDriver that the function is a member of. It can be used by the block driver to access the m_BlockDriverData member or to call the other functions.

sector

This specifies which sector to read.

data

This points to a BYTE array that the block driver needs to fill with the read data.

Return value

Return ECFERR_SUCCESS if the read was successful.

If the read fails, return one of the ECF error codes defined in ECF.h. You can also define your own error codes, error no 64 to 127 are reservered for custom block driver errors.

Remarks

ECF will call this function when it wants to read a sector from the storage device. The fnReadSector function is part of struct ECF_BlockDriver. You need to supply it when writing a block driver.

On both single- and multithreaded systems, ECF will make certain that it will not call any of the other block driver functions until this call has been completed so you do not need to implement any locking in the block driver unless it is needed for other purposes.

Example Code

```
// 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;
}

int main(void)
{
   ...

   struct ECF_BlockDriver bd;

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

   bd.fnReadSector           = RAM_ReadSector;
   bd.fnWriteSector          = RAM_WriteSector;
   bd.fnGetVolumeInformation = RAM_GetVolumeInformation;

   // You can now call ECF_Mount() or ECF_Format() with bd

   ...
}
```

6.1.2 fnWriteSector

fnWriteSector writes a single sector to the block device (usually an SD card or flash memory).

```
ECF_ErrorCode fnWriteSector(
   struct ECF_BlockDriver *bd,
   DWORD sector,
   BYTE *data
);
```

Parameters

bd

This is a pointer to the struct ECF_BlockDriver that the function is a member of. It can be used by the block driver to access the m_BlockDriverData member or to call the other functions.

sector

This specifies which sector to write.

data

This points to a BYTE array with the data for the block driver to write.

Return value

Return ECFERR_SUCCESS if the read was successful.

If the write fails, return one of the ECF error codes defined in ECF.h. You can also define your own error codes, error no 64 to 127 are reservered for custom block driver errors.

Remarks

The fnWriteSector function is part of struct ECF_BlockDriver. You need to supply it when writing a block driver. ECF will call this function when it wants to write a sector to the storage device.

On both single- and multithreaded systems, ECF will make certain that it will not call any of the other block driver functions until this call has been completed so you do not need to implement any locking in the block driver unless it is needed for other purposes.

Example Code

See example for fnReadSector

6.1.3 fnGetVolumeInformation

fnGetVolumeInformation returns information about the disk size and sector size of a block device.

```
ECF_ErrorCode fnGetVolumeInformation(
   struct ECF_BlockDriver *bd,
   WORD *pSectorSize,
   DWORD *pNumberOfSectors
);
```

Parameters

bd

This is a pointer to the struct ECF_BlockDriver that the function is a member of. It can be used by the block driver to access the m_BlockDriverData member or to call the other functions.

pSectorSize

This is a pointer to a WORD that should be set with the sector size.

pNumberOfSectors

This is a pointer to a DWORD that should be set with the number of sectors.

Return value

Return ECFERR_SUCCESS if the read was successful.

If the write fails, return one of the ECF error code defined in ECF.h. You can also define your own error codes, error no 64 to 127 are reservered for custom block driver errors.

Remarks

The fnGetVolumeInformation function is part of struct ECF_BlockDriver. You need to supply it when writing a block driver. ECF will call this function to determine the sector size and how many sectors are there are on a block device.

On both single- and multithreaded systems, ECF will make certain that it will not call any of the other block driver functions until this call has been completed so you do not need to implement any locking in the block driver unless it is needed for other purposes.

Example Code

See example for fnReadSector

6.1.4 fnTrimSectorRange

fnTrimSectorRange trims a range of sectors on the block device. By trimming sectors, ECF signals that these sectors are not used and does not need to be stored.

```
ECF_ErrorCode fnTrimSectorRange(
   struct ECF_BlockDriver *bd,
   DWORD startSector,
   DWORD endSector
);
```

Parameters

bd
This is a pointer to the struct ECF_BlockDriver that the function is a member of. It can be used by the block driver to access the m_BlockDriverData member or to call the other functions.

startSector
This specifies the first sector to trim.

endSector
This specifies the last sector to trim.

Return value

Return ECFERR_SUCCESS if the read was successful.

If the trim fails, return one of the ECF error codes defined in ECF.h. You can also define your own error codes, error no 64 to 127 are reserved for custom block driver errors.

Remarks

The fnTrimSectorRange function is part of struct ECF_BlockDriver. You may supply it when writing a block driver. This function is not mandatory in a block driver and can be NULL.

ECF will call this function to trim sectors. The purpose of trimming is to tell the block device that a sector is no longer in use. Some block drivers benefit from knowning which sectors are in use by e.g. pre-erasing these sectors or by using the unused storage area for something else.

The sector range from and including startSector to and including endSector should be trimmed. That is, fnTrimSectorRange(&bd, 5, 7) trims sectors 5, 6 and 7. fnTrimSectorRange(&bd, 9, 9) trims sector 9.

Since most block drivers will pre-erase a sector when fnTrimSectorRange is called, ECF will try to call fnTrimSectorRange with as large ranges as possible so if the underlying hardware supports erases larger than a sector, it is useful to check the range and see if a more effecient operation can be performed.

On both single- and multithreaded systems, ECF will make certain that it will not call any of the other BlockDriver functions until this call has been completed so you do not need to implement any locking in the block driver unless it is needed for other purposes.

Example Code

```
// Assume a flash chip with 1024 pages where there is a page-erase,
```

```
// a block-erase (16 pages on this chip) and a chip-erase.
ECF_ErrorCode FlashDriver_TrimSectorRange(struct ECF_BlockDriver *bd,
DWORD startSector, DWORD endSector)
{
   DWORD sector;

   if(startSector == 0 && endSector == 1023) {
      EraseFlashChip();
   } else if( (startSector & 0xF) == 0 && (endSector & 0xF) == 0xF) {
      for(sector = startSector;sector <= endSector;sector += 16)
         EraseFlashBlock(sector>>4);
   } else {
      for(sector = startSector;sector <= endSector;sector++)
         EraseFlashSector(sector);
   }

   return ECFERR_SUCCESS;
}
```

6.1.5 fnWriteTrimmedSector

fnWriteTrimmedSector writes a single sector to the block device that has previously been trimmed.

```
ECF_ErrorCode fnWriteTrimmedSector(
   struct ECF_BlockDriver *bd,
   DWORD sector,
   BYTE *data
);
```

Parameters

bd

This is a pointer to the struct ECF_BlockDriver that the function is a member of. It can be used by the block driver to access the m_BlockDriverData member or to call the other functions.

sector

This specifies which sector to write.

data

This points to a BYTE array with the data for the block driver to write.

Return value

Return ECFERR_SUCCESS if the read was successful.

If the write fails, return one of the ECF error code defined in ECF.h. You can also define your own error codes, error no 64 to 127 are reservered for custom block driver errors.

Remarks

ECF will call this function when it wants to write a sector to the storage device and that sector has previously been trimmed. The fnWriteTrimmedSector function is part of struct ECF_BlockDriver. You may supply it when writing a block driver. If fnWriteTrimmedSector is not supplied, fnWriteSector will be called instead, even for sectors that have previously been trimmed.

Please note that although ECF guarantees that it will always call fnTrimSectorRange at some point before calling fnWriteTrimmedSector for a sector, this can not be guaranteed during software or power failures. This means that if you pre-erase sectors when fnTrimSectorRange is called, you should verify that they are actually erased before writing them even if fnWriteTrimmedSector is called.

On both single- and multithreaded systems, ECF will make certain that it will not call any of the other block driver functions until this call has been completed so you do not need to implement any locking in the block driver unless it is needed for other purposes.

Example Code

```
ECF_ErrorCode FlashDriver_WriteSector(struct ECF_BlockDriver *bd,
DWORD sector, BYTE *data)
{
   // Error checking omitted
   EraseFlashSector(sector);
   WriteFlashData(sector, data);
```

```
   return ECFERR_SUCCESS;
}

ECF_ErrorCode FlashDriver_WriteTrimmedSector(struct ECF_BlockDriver
*bd, DWORD sector, BYTE *data)
{
   // Error checking omitted
   if(!CheckIfFlashSectorIsErased(sector))
      EraseFlashSector(sector);

   WriteFlashData(sector, data);

   return ECFERR_SUCCESS;
}
```

6.2 struct ECF_FileHandle

struct ECF_FileHandle is used by the ECF file handling functions to represent a handle to an open file. It is also used by the ECF_ScanDir* functions to keep track of the current directory scan.

```
struct ECF_FileHandle
{
   ...
};
```

Members

The internal members of ECF_FileHandle must not be accessed directly by the user.

See also

ECF_OpenFile(), ECF_ScanDirBegin().

6.3 struct ECF_FileDirectoryData

struct ECF_FileDirectoryData is used by ECF_ScanDirNext() and ECF_GetFileInfo() to return information about a specific file or directory entry.

```
struct ECF_FileDirectoryData
{
   char  m_szFileName[256];
   char  m_szShortFileName[13];
   BYTE  m_dirAttr;
   struct ECF_DateTime m_creationTime;
   struct ECF_DateTime m_lastAccessTime;
   struct ECF_DateTime m_lastWriteTime;
   DWORD m_dirFileSize;
   ...
};
```

Members

m_szFileName:
This is the name of the file or directory. This field contains the long name and is only available if ECF_OPT_SUPPORT_LONG_FILENAMES is defined.

m_szShortFileName:
This is the short name of the file.

m_dirAttr:
The entry's attributes. Valid flags are `ECF_ATTR_READ_ONLY`, `ECF_ATTR_HIDDEN`, `ECF_ATTR_SYSTEM`, `ECF_ATTR_DIRECTORY` and `ECF_ATTR_ARCHIVE`.

m_creationTime:
Timestamp of when the file/directory was created. Has a resolution of 0,01 seconds.

m_lastAccessTime:
Timestamp of when the file/directory was last access. Has a resolution of 1 day. Note: ECF will not update this value when doing file reads to avoid unnecessary writes.

m_lastWriteTime:
Timestamp of when the file/directory was last written. Has a resolution of 2 seconds.

m_dirFileSize:
The size of the file.

6.4 struct ECF_DateTime

struct ECF_DateTime is used to represent time within ECF. But note that the FAT file system does not support the full resolution for all dates and times for all its fields. See comment for each field on what resolution is supported.

```
struct ECF_DateTime
{
   WORD m_wYear;
   BYTE m_bMonth;
   BYTE m_bDay;
   BYTE m_bHour;
   BYTE m_bMinute;
   BYTE m_bSecond;
   BYTE m_bHundredth;
};
```

Members

m_wYear:
The year. E.g. 2012

m_bMonth:
The month. 1 = January, 12 = December.

m_bDay:
The day of the month. 1 – 28,29,30 or 31 depending on which month it is.

m_bHour:
The hour of the day. 0 – 23.

m_bMinute:
Minute. 0 - 59

m_bSecond:
Second. 0 - 59.

m_bHundredth:
Hundredths (1/100 parts) of a second. 0 - 99.

6.4.1 Converting to time_t

You can use the code below to convert from an ECF_DateTime into a time_t (with epoch of 1970-01-01 00:00:00)

```
// Will convert a struct ECF_DateTime to time_t (seconds since epoch
// 1970-01-01 00:00:00)
// Valid for times between 1970 and 2037 for a signed 32-bit time_t
// Valid for times between 1970 and 2099 for an unsigned 32-bit time_t
// Hundredths will be lost in the conversion
ECF_ErrorCode ECF_DateTimeToTimeT(struct ECF_DateTime *dateTime,
time_t *t)
{
   const BYTE bDaysInMonth[13] =
      {31,28,31,30,31,30,31,31,30,31,30,255};
   WORD days;
   BYTE month;
   BYTE year;
   BOOL isLeapYear;

   if(dateTime->m_wYear < 1970 ||
      dateTime->m_wYear >= 2100 ||
      dateTime->m_bMonth > 12 ||
      dateTime->m_bMonth < 1 ||
      dateTime->m_bDay < 1 ||
      dateTime->m_bHour > 23 ||
      dateTime->m_bMinute > 59 ||
      dateTime->m_bSecond > 59)
         goto exit_function_and_fail;

   year = dateTime->m_wYear-1970;
   isLeapYear = (((year+2) & 3) == 0);
   days = (year*365 + (year+1)/4);

   for(month = 0;month < (dateTime->m_bMonth-1);month++)
   {
      days += bDaysInMonth[month];
      if(month == 1 && isLeapYear) // February on a leap year
         days++;
   }
   days += dateTime->m_bDay-1;
   if((dateTime->m_bDay-1) >= bDaysInMonth[dateTime->m_bMonth-1])
   {
      if(dateTime->m_bMonth != 2 &&
         dateTime->m_bDay == 29 &&
         !isLeapYear)
         goto exit_function_and_fail;
   }

   *t = (time_t)((time_t)days*86400 + dateTime->m_bHour*3600 +
         dateTime->m_bMinute*60 + dateTime->m_bSecond);

   return ECFERR_SUCCESS;
exit_function_and_fail:
   *t = 0;
   return ECFERR_PARAMETERERROR;
}
```

6.4.2 Converting from time_t

You can use the code below to convert from an a time_t (with epoch of 1970-01-01 00:00:00) into an ECF_DateTime

```
// Will convert a struct ECF_DateTime to time_t (seconds since epoch
1970-01-01 00:00:00)
// Valid for times between 1970 and 2037 for a signed 32-bit time_t
// Valid for times between 1970 and 2099 for an unsigned 32-bit time_t
ECF_ErrorCode ECF_TimeTToDateTime(time_t t, struct ECF_DateTime
*dateTime)
{
   const BYTE bDaysInMonth[13] =
      {31,28,31,30,31,30,31,31,30,31,30,255};
   WORD days = (WORD)(t / 86400);
   DWORD secondsInDay = (DWORD)t % 86400;
   BYTE month = 0;
   BYTE daysInFebruary = 28;
   BYTE year;

   year = (days - ((days / 365)+1)/4) / 365;
   days -= year*365 + (year+1)/4;

   if(((year+2) & 3) == 0)
      daysInFebruary++;

   if(days >= 31) // January
   {
      month++;
      days -= 31;
      if(days >= daysInFebruary) { // February
         month++;
         days -= daysInFebruary;

         while(days >= bDaysInMonth[month])
            days -= bDaysInMonth[month++];
      }
   }

   dateTime->m_wYear = year+1970;
   dateTime->m_bMonth = month+1;
   dateTime->m_bDay = (BYTE)days+1;
   dateTime->m_bHour = (BYTE)(secondsInDay/3600);
   dateTime->m_bMinute = (BYTE)((secondsInDay/60) % 60);
   dateTime->m_bSecond = (BYTE)(secondsInDay % 60);
   dateTime->m_bHundredth = 0;

   return ECFERR_SUCCESS;
}
```

7 Options (defines)

ECF will include a file called Project.h. You should make all your ECF defines in Project.h.

7.1 ECF_OPT_SUPPORT_ALL_SECTORSIZES

Define to support all possible sector sizes. If defined, 512, 1024, 2048 and 4096 will supported as sector sizes. If not, the only supported sector size will be 512.

Most devices uses a 512 bytes sector size but if you need to support unknown devices you need to define ECF_OPT_SUPPORT_ALL_SECTORSIZES. But by defining it, ECF and the cache will use more memory.

7.2 ECF_OPT_SUPPORTED_MOUNTPOINTS

Defines how many drives can be mounted. If not defined, a default of 1 will be used. Values between 1 and 26 are supported.

7.3 ECF_OPT_SUPPORT_FORMAT

Define if you need ECF_Format(), ECF_CreatePartitionTable() and/or ECF_CreatePartition().

7.4 ECF_OPT_SUPPORT_LONG_FILENAMES

Define if you want to support long file names.

7.5 ECF_OPT_SECTOR_CACHE

Define how many sectors will be held in the cache. Recommended value is 4 or above but you can set it to 1 if you need to save memory and don't mind a slower speed.

7.6 ECF_OPT_ATTEMPT_ORDERED_WRITE

Define to make ECF flush its cache in order rather than by usage. Useful for devices that internally have bigger page sizes than a sector and benefits from having the sectors written in order.

7.7 ECF_OPT_USE_MUTEX

Define to use a mutex to lock the file system. You will also need to define:

```
ECF_OPT_MUTEX_TYPE
ECF_OPT_MUTEX_INIT(m)
ECF_OPT_MUTEX_ACQUIRE(m)
ECF_OPT_MUTEX_RELEASE(m)
ECF_OPT_MUTEX_CLEANUP(m)
```

Example for FreeRTOS:

In Project.h:

```
#define ECF_OPT_USE_MUTEX
```

```
#define ECF_OPT_MUTEX_TYPE          void *
#define ECF_OPT_MUTEX_INIT(m)       Mutex_Init(&m)
#define ECF_OPT_MUTEX_ACQUIRE(m)    Mutex_Acquire(m)
#define ECF_OPT_MUTEX_RELEASE(m)    Mutex_Release(m)
#define ECF_OPT_MUTEX_CLEANUP(m)    Mutex_CleanUp(m)
```

Somewhere in a .c file:

```
BOOL Mutex_Init(void **m)
{
   vSemaphoreCreateBinary(*m);
   return TRUE;
}

void Mutex_Acquire(void *m)
{
   xSemaphoreTake(m, portMAX_DELAY);
}

void Mutex_Release(void *m)
{
   xSemaphoreGive(m);
}

void Mutex_CleanUp(void *m)
{
   vSemaphoreDelete(m);
}
```

7.8 ECF_OPT_PROGRESS_CALLBACK

Set to define a progress callback for lengthy operations. Currently only used by ECF_Format()

7.9 ECF_OPT_WATCHDOG_CALLBACK

Will be called when ECF performs a lengthy operation about once for every block driver operation.

Please note the following:

- Although ECF will reset the watchdog during lengthy operations, you will still need to occasionally reset it in your other code that does not call ECF.
- If you are using a very large cache, ECF might not need to call the block driver and the watchdog will not be reset. But since all operations are in the cache, they should be fast and ECF should return quickly.
- Although most blockdriver operations are relatively quick in comparison to the watchdog timer, fnTrimSectorRange () might be called with a wide range and may take a lot of time to complete. So you might need to add a watchdog reset in your fnTrimSectorRange() function.

7.10 ECF_OPT_CURRENT_TIME_FUNCTION

Define to make ECF aware of time and write appropriate time stamps.

```
#define ECF_OPT_CURRENT_TIME_FUNCTION(ecfdatetime) \
             Runtime_RetrieveDateTime(ecfdatetime)
```

Somewhere in a .c file:

```
void Runtime_RetrieveDateTime(struct ECF_DateTime *dateTime)
{
   int year, month, day, hour, minute, second, milliseconds;

   // TODO: Retreive year, month, day, hour, minute, second
   // and milliseconds

   dateTime.m_wYear = year; // I.e 2012
   dateTime.m_bMonth = month;  // 1 = January, 12 = December
   dateTime.m_bDay = day; // Day of the month 1-28,29,30,31
   dateTime.m_bHour = hour; // Hour of the day 0 - 23
   dateTime.m_bMinute = minute; // Minute 0-59
   dateTime.m_bSeconds = seconds; // Seconds 0-59
   dateTime.m_bHundredth = milliseconds/10;
}
```

Note: The function you define is supposed to return quickly so if you have real time clock (RTC) that is slow to access, you should periodically check your RTC, store the values in RAM and return these values then the function is called.