high integrity systems

CONNECT Middleware User Manual

FAT 16/ FAT 32 File Systems

Version 7.5, 09 May 2017
Disclaimer

The content provided in this document is believed to be accurate, but not guaranteed to be entirely free from errors. The content of this document is also subject to change. While the information herein is assumed to be accurate, WITTENSTEIN high integrity systems accepts no liability whatsoever for errors and omissions, and assumes that all users understand the risks involved.

Copyright notice

None of the materials provided in this document may be used, reproduced or transmitted, in any form or by any means without written permission from WITTENSTEIN high integrity systems.

All rights reserved.

© 2017 WITTENSTEIN aerospace & simulation Ltd

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective companies.

Contact Address

WITTENSTEIN high integrity systems
Brown’s Court, Yanley Lane
Long Ashton Business Park
Long Ashton
Bristol
BS41 9LB
ENGLAND, UK

www.HighIntegritySystems.com
sales@HighIntegritySystems.com
Americas:  +1 408 625 4712
ROTW  +44 1275 395 600

WITTENSTEIN high integrity systems is a trading name of WITTENSTEIN aerospace & simulation ltd.
About WITTENSTEIN high integrity systems

WITTENSTEIN high integrity systems (WHIS) is a safety systems company that produces and supplies real time operating systems (RTOS) and software components to the medical, aerospace, transportation and industrial sectors. WHIS is part of the WITTENSTEIN Group, a global technology company established in 1948 with a presence in over 45 countries.

Relationship with FreeRTOS

WHIS leverage RTOS technology from the FreeRTOS project, the market leading embedded real time operating system from Real Time Engineers, download in excess of 100,000 times in 2013.

WHIS have a unique relationship with the FreeRTOS project as Richard Barry, the creator of FreeRTOS and the owner of Real Time Engineers Ltd, is also the Innovation Leader for WHIS and was instrumental in setting up the division.

OPENRTOS

OPENRTOS provides the only available commercial license for FreeRTOS, the highly successful, small, efficient embedded real time operating system distributed under a modified GPL license. OPENRTOS and FreeRTOS share the same code base, however OPENRTOS truly transitions developers into the professional world, with full commercial licensing and access to direct support, backed up by tools, training and consultancy services. Developers can extend the functionality of OPENRTOS by selecting from a wide range of middleware components and Board Support Packages.

SAFERTOS

SAFERTOS is a pre-emptive, pre-certified real time operating system that delivers unprecedented levels of determinism and robustness to embedded systems. Based on the FreeRTOS function model, but specifically re-designed for the safety market by our own team of safety experts, SAFERTOS is independently certified by TUV SUD to IEC 61508 SIL3.

CONNECT Middleware

CONNECT MIDDLEWARE components are feature rich and designed specifically for embedded platforms. They are available with all WHIS RTOS products as one highly integrated, fully optimized and verified package accompanied by a demonstration application.

CONNECT MIDDLEWARE supports USB Host &Device, Networking and File systems.
# Table of Contents

1. Introduction .............................................................................................................. 7

2. Microsoft FAT File System Overview ...................................................................... 8

3. Architecture ............................................................................................................... 9
   3.1 File System API ....................................................................................................... 9
   3.2 FAT16/FAT32 File System ...................................................................................... 9
   3.3 Storage Media Driver ............................................................................................ 10

4. File System POSIX API ............................................................................................ 11
   4.1 Open a file: fopen() ............................................................................................. 13
   4.2 Read data from file: fread() .................................................................................. 14
   4.3 Write data to file: fwrite() ................................................................................... 15
   4.4 Close file: fclose() ............................................................................................... 17
   4.5 Set Position: fseek() ........................................................................................... 18
   4.6 Get Current Position: ftell() ................................................................................ 19
   4.7 Set Position: fsetpos() .......................................................................................... 20
   4.8 Get Current Position: fgetpos() ............................................................................ 21
   4.9 Read Next Character: fgetc() ............................................................................... 22
   4.10 Read String: fgets() ............................................................................................ 23
   4.11 Write a Character: fputc() .................................................................................. 24
   4.12 Write String: fputs() ......................................................................................... 25
   4.13 Reports end of file: feof() ................................................................................... 26
   4.14 Gives error: ferror() ............................................................................................ 27
4.15 Formats Volume: fformat() ........................................................................................................... 28
4.16 Delete a file: fdelete() .................................................................................................................. 29
4.17 Initializing File System: finit () ...................................................................................................... 30
4.18 Rename a File or Directory: frename() .......................................................................................... 31
4.19 Move a File or Directory: fmove() .................................................................................................. 32
4.20 Create a Directory: fmkdir () ........................................................................................................ 33
4.21 Remove a Directory: frmdir () ...................................................................................................... 34
4.22 Gives free space: ffree () ................................................................................................................ 35
4.23 Search files and directories: ffind () .............................................................................................. 36
4.24 Search next file or directory: ffindnext () ..................................................................................... 37
4.25 Close opened path for search: ffindclose () .................................................................................. 38
4.26 Writes formatted text to File: fprintf () ....................................................................................... 39
5. Volume Status Notify Callback Function ......................................................................................... 40
6. Journaling .......................................................................................................................................... 42
   6.1 How Journaling Works ................................................................................................................... 42
7. Storage Driver .................................................................................................................................. 43
   7.1 Initialize Storage: xStoreInit() ...................................................................................................... 44
   7.2 Read capacity: xStoreCapacityRead() .......................................................................................... 45
   7.3 Read data from Storage: xStoreRead() .......................................................................................... 46
   7.4 Write Data to Storage: xStoreWrite() ........................................................................................... 47
   7.5 Read status: xStoreStatusRead() .................................................................................................. 48
   7.6 Read configuration: xStoreConfigurationRead() ........................................................................ 49
   7.6.1 Storage Configuration: StoreConfiguration_t ............................................................................. 49
8. Storage Device API ........................................................................................................................................52

8.1 Initialize storage driver: xstoreStorageDriverInit() ..................................................................................52

8.2 Add storage device to file system: xfiledevDeviceAdd() ..............................................................................53

8.3 Remove storage device from file system: vfiledevDeviceRemove() ..........................................................53

9. CONNECT File System Example ..................................................................................................................55

Appendix A: Error Codes ..................................................................................................................................57
1. Introduction

The CONNECT File System is a compact and highly reliable embedded FAT16/FAT32 File System, designed for embedded applications requiring data storage to physical media such as SD/MMC, USB and FLASH.

The File System API layer contains two interfaces. One is a POSIX interface that provides standard file system API calls such as fopen(), fread(), fwrite() and fclose() etc. The second is a Native interface that provides non-standard API calls for operations such as directory and file system management.

CONNECT File System is delivered with a highly optimized integration for either FreeRTOS/OPENRTOS for commercial products, or SAFERTOS, for products requiring certification to international safety standards such as IEC 61508.

CONNECT File System is supplied with full source code, distributed under a straightforward perpetual license, with no runtime fees or royalties. Customers receive comprehensive documentation and our full support.

Features

- Compact & full featured embedded File System
- Supports RAM, NAND, NOR, SD, MMC and USB Mass storage devices
- MS DOS/MS Windows compatible FAT16/32 file system
- FLASH Translation Layer providing, Wear Leveling, Bad Block Management, ECC and Garbage Collection
- Multiple logical volumes and storage devices
- Full C source code supplied
- Close integration with FreeRTOS, OPENRTOS and SAFERTOS
- Delivers high levels of data throughput, whilst utilizing minimum system resources
- Long file names
- Easily integrated with virtually any FLASH or physical media device.
2. Microsoft FAT File System Overview

This section provides a brief overview about the Microsoft FAT File System. For a complete, in-depth description, please refer the Microsoft EFI FAT32 File System Specification by clicking on the link below.

http://www.microsoft.com/whdc/system/platform/firmware/fatgendown.mspx

FAT is a file system specification developed by Microsoft. There are three types of FAT File Systems FAT32, FAT16 and FAT12.

FAT12 was the initial version, introduced in 1977 before the launch of MS-DOS. FAT12 was the primary file system for Microsoft operating systems up to MS-DOS4.0. FAT12 supports drive sizes up to 32MB.

FAT16 was the second version released, introduced in 1988. FAT16 was the primary file system for MS-DOS4.0 through to Microsoft Windows95. FAT16 supports drive sizes up to 2GB.

FAT32 is the latest version of the FAT file system, introduced in 1996 for Windows 95 OSR2 users and was the primary file system for consumer windows versions through Windows ME. FAT32 supports drive sizes up to 8TB.
3. Architecture

The CONNECT File System architecture has a modular design allowing it to support multiple different storage I/O devices. The File System has a tight integration with FreeRTOS, OPENRTOS or SAFERTOS.

Figure 3-1 shows a block diagram of the different layers of the CONNECT File System.

![CONNECT File System Architecture Diagram](image)

**FIGURE 3-1 CONNECT FILE SYSTEM ARCHITECTURE DIAGRAM**

3.1 File System API

The File System API layer provides two interfaces to the application. One is a POSIX interface that provides standard file system API calls such as fopen(), fread(), fwrite() and fclose() etc. The second is a Native interface that provides non-standard API calls for operations like directory and file system management.

3.2 FAT16/FAT32 File System

The file system layer implements FAT16/FAT32 protocol. This layer translates the file system operations to block I/O requests and forwards them to the corresponding Storage Media Driver. This layer also manages dynamic attachment/removal of storage devices.
3.3 Storage Media Driver

The Storage Media Drivers are hardware dependent and provide all the low-level functionality required by the File System for accessing NAND/NOR/SD/MMC and RAM Disk devices.
## 4. File System POSIX API

### TABLE 4-1 FILE SYSTEM POSIX API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen()</td>
<td>Open the given file.</td>
</tr>
<tr>
<td>fread()</td>
<td>Read the number of requested bytes from an opened file.</td>
</tr>
<tr>
<td>fwrite()</td>
<td>Write data to an opened file.</td>
</tr>
<tr>
<td>fclose()</td>
<td>Close an already opened file.</td>
</tr>
<tr>
<td>fseek()</td>
<td>Set the cursor at required position.</td>
</tr>
<tr>
<td>ftell()</td>
<td>Tell file offset position.</td>
</tr>
<tr>
<td>fsetpos()</td>
<td>Similar to fseek()</td>
</tr>
<tr>
<td>fgetpos()</td>
<td>Similar to ftell()</td>
</tr>
<tr>
<td>fgetc()</td>
<td>Get character from opened file.</td>
</tr>
<tr>
<td>fgets()</td>
<td>Get string from opened file.</td>
</tr>
<tr>
<td>fputc()</td>
<td>Write character to opened file.</td>
</tr>
<tr>
<td>fputs()</td>
<td>Write string to opened file.</td>
</tr>
<tr>
<td>feof()</td>
<td>Reports whether end of file has been reached or not.</td>
</tr>
<tr>
<td>ferror()</td>
<td>Reports the error status relating to recent read/write operation.</td>
</tr>
<tr>
<td>fformat()</td>
<td>Formats given volume in specified device.</td>
</tr>
<tr>
<td>fdelete()</td>
<td>Deletes the file in given path.</td>
</tr>
<tr>
<td>finit()</td>
<td>Initializes the file system.</td>
</tr>
<tr>
<td>rename()</td>
<td>Renames a file or directory.</td>
</tr>
<tr>
<td>fmove()</td>
<td>Moves a file or directory.</td>
</tr>
<tr>
<td>fmkdir()</td>
<td>Creates a directory.</td>
</tr>
<tr>
<td>frmdir()</td>
<td>Removes a directory.</td>
</tr>
<tr>
<td>ffree()</td>
<td>Gives free space on specified volume.</td>
</tr>
<tr>
<td>ffind()</td>
<td>Searches specified pattern files or directories in given path.</td>
</tr>
<tr>
<td>fffindnext()</td>
<td>Searches next specified pattern file or directory.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>ffindclose()</td>
<td>Close already opened path for searching.</td>
</tr>
<tr>
<td>fprintf()</td>
<td>Writes formatted text to file.</td>
</tr>
</tbody>
</table>
Open a file: fopen()

This function opens a file.

```c
FILE *fopen ( const char *pcFilePath,
               const char *pcFileMode );
```

Arguments

- `pcFilePath` Pointer to the absolute path of a file.
- `pcFileMode` The mode determines the type of access permitted on the file as detailed in the table below.

### TABLE 4-2 PERMITTED PMODE VALUES

<table>
<thead>
<tr>
<th>PMODE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;rb&quot;</td>
<td>Opens a file in binary mode for reading. The fopen call fails if the file does not exist.</td>
</tr>
<tr>
<td>&quot;wb&quot;</td>
<td>Opens an empty file in binary mode for writing. If the given file already exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;ab&quot;</td>
<td>Opens a file in binary mode for writing at the end of file.</td>
</tr>
<tr>
<td>&quot;rb+&quot;</td>
<td>Opens an existing file in binary mode for reading and writing.</td>
</tr>
<tr>
<td>&quot;wb+&quot;</td>
<td>Opens an empty file in binary mode for reading and writing. If the given file already exists, its contents are destroyed.</td>
</tr>
<tr>
<td>&quot;ab+&quot;</td>
<td>Opens a file for reading and appending</td>
</tr>
</tbody>
</table>

Return Value

On success, returns a pointer to the open file. Otherwise returns NULL pointer.

Example

```c
FILE *fp;
fp = fopen( “usb0:/test.txt” , ”wb” );
```
4.2 Read data from file: fread()

This function reads data from opened file.

```c
#include <stdio.h>

int fread ( void *pcBuffer,
            size_t xBlockSize,
            size_t xCount,
            FILE *pxFilePointer );
```

Arguments

- `pcBuffer`: Pointer to a buffer to store the data read from file.
- `xBlockSize`: Item size in bytes.
- `xCount`: Number of items to read.
- `pxFilePointer`: Pointer to FILE structure.

Return Value

Returns the number of items actually read. The number of items read may be less than `xCount` if an end of the file is encountered before `xCount` items have been read or if an error occurs during the read operation.

Example

```c
FILE *fp;
int read;
char buf[200];

fp = fopen( “usb0:/test.txt”, ”rb” );
read = fread( ( void * )buf , 100 , 2 , fp );
```
4.3  Write data to file: fwrite()

This function writes data to an opened file.

```c
int fwrite( void *pvBuffer,
            size_t xBlockSize,
            size_t xCount,
            FILE *pxFilePointer);
```

**Arguments**

- **pvBuffer**  
  Pointer to a buffer containing data to be written to file.

- **xBlockSize**  
  Item size in bytes.

- **xCount**  
  Number of items to write.

- **pxFilePointer**  
  Pointer to FILE structure.

**Return Value**

Returns the number of items actually written to file. The number of items written may be less than `xCount` if error occurs during the write operation.

**Configuration**

The operation of the fwrite function depends on the setting of the filecfgOVER_WRITE_DATA_OPTION macro located in file_cfg.h. This macro has two possible values.
### TABLE 4-3 FWRITE CONFIGURATION OPTIONS

<table>
<thead>
<tr>
<th>Configuration Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filecfgOPTION_ALLOC_FREE_AND_WRITE</td>
<td>Allocates free clusters and writes new data and then deletes old clusters.</td>
</tr>
<tr>
<td></td>
<td>If journaling enabled, it will recover the file system if a crash happens during the write operation.</td>
</tr>
<tr>
<td>filecfgOPTION_DIRECTLY_OVER_WRITE</td>
<td>Directly overwrites old clusters with new data.</td>
</tr>
<tr>
<td></td>
<td>Even journaling enabled, if new clusters are required when writing data then only journal recovers file system if any crash happens during the write operation.</td>
</tr>
</tbody>
</table>

**Example**

```c
FILE *fp;
int written;
char buf[200];

fp = fopen("usb0:/test.txt", "wb");
written = fwrite((void*)buf, 100, 2, fp);
```
4.4 Close file: fclose()

This function closes the file.

\[
\text{int fclose ( FILE *pxFilePointer );}
\]

**Arguments**

pxFilePointer Pointer to FILE structure.

**Return Value**

If successful, this function returns zero. On a failure condition, this function returns an error code.

**Example**

```c
FILE *fp;

fp = fopen( "usb0:/test.txt" , "rb" );
fclose( fp );
```
4.5 Set Position: fseek()

This function moves the file pointer to a specified position in a file.

```c
int fseek ( FILE *pxFilePointer, long lBytesToSeek, int lOrigin );
```

**Arguments**

- `pxFilePointer` Pointer to FILE structure.
- `lBytesToSeek` Number of bytes from origin.
- `lOrigin` The starting point within the file, as detailed on Table 4-4.

**TABLE 4-4 PERMITTED ORIGIN VALUES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filefatSEEK_SET 0</td>
<td>Beginning of file</td>
</tr>
<tr>
<td>filefatSEEK_CUR 1</td>
<td>Current position</td>
</tr>
<tr>
<td>filefatSEEK_END 2</td>
<td>End of file</td>
</tr>
</tbody>
</table>

**Return Value**

If successful this function returns zero. On failure this function returns an error code.

**Example**

```c
FILE *fp;

fp = fopen( "usb0:/test.txt" , "rb" );
fseek( fp, 10, filefatSEEK_SET );
```
4.6  Get Current Position: ftell()

This function gets the current position of a file pointer.

\[
\text{long ftell ( FILE *pxFilePointer );}
\]

**Arguments**

*pxFilePointer  
Pointer to FILE structure

**Return Value**

On success, returns the current position of the file pointer, otherwise an error code is returned.

**Example**

```c
FILE *fp;
long curpos;

fp = fopen( "usb0:/test.txt", "rb" );
curpos = ftell( fp );
```
4.7 Set Position: fsetpos()

This function sets the position of a file pointer.

```c
int fsetpos ( FILE *pxFilePointer,
                 const fpos_t *pxPosition );
```

**Arguments**

- `pxFilePointer` Pointer to FILE structure
- `pxPosition` Pointer containing the file position

**Return Value**

On success this function returns zero. On failure this function returns error code.

**Example**

```c
FILE *fp;

fpos_t pos = 23;

fp = fopen("usb0:/test.txt", "rb");

fsetpos(fp, &pos);
```
4.8 Get Current Position: fgetpos()

This function gets the current position of a file pointer.

```c
int fgetpos ( FILE *pxFilePointer
              , fpos_t *pxPosition )
```

**Arguments**

- `pxFilePointer` Pointer to FILE structure
- `pxPosition` Pointer to store the file position

**Return Value**

On success this function returns zero. On failure this function returns error code.

**Example**

```c
FILE  *fp;
fpos_t pos;

fp = fopen( "usb0:/test.txt", "rb" );
fgetpos(fp, &pos);
```
4.9 Read Next Character: fgetc()

This function reads a character from the current position.

```c
int fgetc ( FILE *pxFilePointer );
```

**Arguments**

pxFilePointer Pointer to FILE structure.

**Return Value**

On success, this function returns a character. Otherwise this function returns -1.

**Example**

```c
FILE *fp;
char ch;

fp = fopen( "usb0:/test.txt" , "rt" );
ch = fgetc( fp );
```
4.10 Read String: fgets()

This function reads num-1 characters from the file and stores them in the given string. This function will terminate early if it reaches end of line character before the read operation is completed. The string parameter is terminated by null character.

```c
char *fgets ( char *pcString,
             int iNumberOfCharacters,
             FILE *pxFilePointer );
```

**Arguments**

pcString Pointer to a string read from file.

iNumberOfCharacters Number of characters to read.

pxFilePointer Pointer to FILE structure.

**Return Value**

On success this function returns string, otherwise a null value is return.

**Example**

```c
FILE *fp;
char str[10], *ptr;

fp = fopen( "usb0:/test.txt" , "rt" );
ptr = fgets( str , 10 , fp );
```
4.11 Write a Character: fputc()

This function writes a single character at current position in the file.

```c
int fputc ( int iCharacter,
            FILE *pxFilePointer);
```

**Arguments**

- `iCharacter` ASCII character to be written.
- `pxFilePointer` Pointer to FILE structure.

**Return Value**

On success, returns the character written. On failure returns -1.

**Example**

```c
FILE *fp;
char ch = 'A';
int result;

fp = fopen( "usb0:/test.txt" , "wt" );
result = fputc( ch, fp );
```
4.12 Write String: fputs()

This function writes a string at current position of file.

```c
int fputs ( const char *pcString,
            FILE *pxFilePointer );
```

**Arguments**

- **pcString**: ASCII string to be written.
- **pxFilePointer**: Pointer to FILE structure.

**Return Value**

On success, returns number of bytes written. On failure, this function returns -1.

**Example**

```c
FILE *fp;
char *str = "Hello World";
int result;

fp = fopen( "usb0:/test.txt" , "wt" );
result = fputs( str, fp );
```
4.13 Reports end of file: `feof()`

This function reports if the end of the file has been reached.

```c
int feof(FILE *pxFilePointer);
```

**Arguments**

pxFilePointer Pointer to FILE structure

**Return Value**

On reaching end of file, returns non-zero, otherwise returns 0.

**Example**

```c
FILE *fp;
int result;

fp = fopen("usb0:/test.txt", "rb");
result = feof(fp);
```
4.14  **Gives error: ferror ( )**

This function reports error events relating to recent read/write operation.

```c
int ferror ( FILE *pxFilePointer );
```

**Arguments**

pxFilePointer  Pointer to FILE structure

**Return Value**

Returns zero on successful read/write operation, otherwise returns an error code.

**Example**

```c
FILE   *fp;
int     read, error;
char   buf[100];

fp    = fopen("usb0:/test.txt", "rb");
read = fread((void *) buf, 100, 1, fp);
error = ferror( fp );
```
4.15 Formats Volume: fformat()

This function formats the given volume with specified format type and assigns the given volume name.

```c
int fformat ( const char *pcDrivePath,
              const char *pcNewVolumeName,
              const char *pcFormatType );
```

**Arguments**

- **pcDrivePath**  Drive path.
- **pcNewVolumeName**, New volume name, put 0 for default volume name.
- **pcFormatType**  Format type, put 0 for default format type.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;

result = fformat( “usb0:/” , “MYVOL” , “FAT16” );
```
4.16 Delete a file: fdelete()

This function deletes a given file.

```c
int fdelete (const char *pcFilePath);
```

**Arguments**

pcFilePath File name to delete.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;
result = fdelete(“usb0:/test.txt”);
```
4.17 Initializing File System: finit ()

This function initializes the file system. See section 5 for details about volume status notify function.

\[
\text{int \hspace{1em}} \text{finit} \hspace{1em} ( \text{filecoreVolumeNotifyFunction_t \hspace{1em}} \text{pxVolumeStatusNotifyFunction} );
\]

**Arguments**

pxVolumeStatusNotifyFunction Function which should be called when status changed for a volume or zero if no status notifications required.

**Return Value**

Returns zero on successful file system initialization, otherwise returns error number.

**Example**

```c
int \hspace{1em} \text{result};
\text{result} = \text{finit}(0);
```
4.18 Rename a File or Directory: rename()

This function renames a file or directory in given path with the specified new name.

```c
int rename ( const char *pcFilePath
              const char *pcNewName );
```

**Arguments**

- `pcFilePath` File or directory path.
- `pcNewName` New file or directory name.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;
result = rename( “usb0:/Dir/test.txt” , “mytest.txt” );
```
4.19 Move a File or Directory: fmove()

This function moves a file or directory in given source path to specified destination path.

```c
int fmove ( const char *pcSourcePath,
            const char *pcDestinationPath );
```

**Arguments**

- `pcSourcePath` Source file or directory path.
- `pcDestinationPath` Destination file or directory path.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;

result = fmove( "usb0:/Dir/test.txt", "usb0:/mytest.txt" );
```
4.20 Create a Directory: fmkdir()

This function creates a directory with the given path.

```c
int fmkdir ( const char *pcDirectoryPath );
```

**Arguments**

pcDirectoryPath  Path to create a new directory.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;

result = fmkdir("usb0:/Dir");
```
4.21  Remove a Directory: frmdir()

This function removes a directory in the given path depending upon the flag value.

```c
int frmdir ( const char *pcDirectoryPath,
             unsigned char ucFlag );
```

**Arguments**

- `pcDirectoryPath` Path to deleted directory.
- `ucFlag` Deletion type flag, for details see the table below.

**TABLE 4-5 PERMITTED DELETION FLAG VALUES**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filefatDIR_DELETE_IF_EMPTY</code> or 0</td>
<td>Deletes only if directory is empty.</td>
</tr>
<tr>
<td><code>fatfileDIR_DELETE_EVEN_NOT_EMPTY</code> or 1</td>
<td>Deletes even directory has contents.</td>
</tr>
</tbody>
</table>

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
int result;
result = frmdir( "usb0:/Dir", filefatDIR_DELETE_IF_EMPTY );
```
4.22  Gives free space: ffree ()

This function gives free space on specified volume.

```c
unsigned int ffree ( const char *pcDiskPath );
```

**Arguments**

pcDiskPath  Disk path.

**Return Value**

On success, returns the freed amount of space. On failure this function returns 0.

**Example**

```c
unsigned int space;

space = ffree( “usb0:/” );
```
4.23  Search files and directories: ffind ()

This function searches specified pattern files or directories in given path.

```c
FileFind_t  *ffind    ( const char  *pcSearchPatternWithPath,
                        FileInformation_t *pxFileInformation );
```

Arguments

- `pcSearchPatternWithPath`: File path and search pattern. Search pattern should be one the following values show in Table 4-6.
- `pxFileInformation`: On success, this File Information structure contains details of first occurrence of search pattern.

<table>
<thead>
<tr>
<th>PERMITTED FLAG VALUE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>* or <em>..</em></td>
<td>Search all files in the directory</td>
</tr>
<tr>
<td>abc*</td>
<td>Search for files that begin with “abc”, e.g. abcd.txt, abc.txt, abcx.</td>
</tr>
<tr>
<td>abc.</td>
<td>Search for files with name “abc” and any extension, e.g. abc.txt, abc.jpg</td>
</tr>
<tr>
<td>*.abc</td>
<td>Search for files contains extension “abc”, e.g. test.abc, ram.abc</td>
</tr>
<tr>
<td>abc*.xyz</td>
<td>Search for files that begin with “abc” and contains extension “xyz”, e.g. abc.xyz, abcd.xyz</td>
</tr>
</tbody>
</table>

Return Value

On success, returns FileFind_t structure. On failure this function returns 0.

Example

```c
FileFind_t  *find;
FileInformation_t info;

find = ffind( “usb0:/*:.*”, &info );
```
4.24 Search next file or directory: ffindnext()

This function searches next specified pattern file or directory from given FileFind_t structure.

```c
int ffindnext ( FileFind_t *pxFileFind,
                FileInformation_t *pxFileInformation );
```

**Arguments**

- pxFileFind: Previous FileFind_t structure as input.
- pxFileInformation: File Information structure as output.

**Return Value**

On success, returns zero. On failure, this function returns error number.

**Example**

```c
FileFind_t *pxFileFind;
FileInformation_t xFileInfo;
int result;

pxFileFind = ffind( "usb0:/.*", &xFileInfo );
result = ffindnext( find, &info );
```
4.25 Close opened path for search: ffindclose()

This function closes the already opened path for searching with ffind(). This must be used like fclose(), otherwise result in unknown operation.

```c
int ffindclose ( FileFind_t *pxFileFind );
```

**Arguments**

pxFileFind Previous FileFind_t structure as input.

**Return Value**

On success, returns zero. On failure this function returns error number.

**Example**

```c
FileFind_t *pxFileFind;
FileInformation_t xFileInfo;
int result;

pxFileFind = ffind( "usb0:/*." , &xFileInfo );
result = ffindclose( pxFileFind );
```
4.26  Writes formatted text to File: fprintf()

This function writes formatted text at current position of file.

```c
int fprintf ( FILE *pxFilePointer,
              const char *pcString
              ... ) ;
```

Arguments

pxFilePointer  Pointer to FILE structure.

pcString      Format string to be written to the file.

Return Value

Number of characters written to file.

Example

```c
FILE *fp;
int written;

fp = fopen( "usb0:/test.txt" );
written = fprintf( fp, "%s %d", "Test Data", 0x1234 );
```
5. Volume Status Notify Callback Function

The volume status notify callback function should be passed as an argument to `finit()` to get status of disk partitions when storage device added or removed.

The following callback function should be implemented by the application to receive volume status when storage is added to file system.

```c
void vFileVolumeNotify ( char *pcVolumePath,
                         uint8_t ucFormatType,
                         FileError_t xMountError,
                         BoolType_t blsPresent );
```

**Arguments**

- **pcVolumePath**  Contains the volume path. Application must use this path to access file system on this volume. The volume path contains storage device name and volume number. Volume numbers start from zero.

  The format of volume path is  DeviceName:[VolumeNumber]/

  Zero is default volume number if not specified in path.

  Examples: USB:/ or USB:1/

- **ucFormatType**  Contains the partition format type. See **TABLE 5-1** for details

- **xMountError**  Contains the error that occurred during the disk partition mount or unmount operation. See **TABLE 5-2** for details.

- **blsPresent**  Contains the volume present status. See **TABLE 5-3** for details

**TABLE 5-1 PARTITION FORMAT TYPES**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unrecognized format</td>
</tr>
<tr>
<td>1 or filefatFAT_12</td>
<td>FAT 12</td>
</tr>
<tr>
<td>2 or filefatFAT_16</td>
<td>FAT 16</td>
</tr>
<tr>
<td>3 or filefatFAT_32</td>
<td>FAT 32</td>
</tr>
</tbody>
</table>
TABLE 5-2 ERRORS THAT CAN OCCUR DURING DISK MOUNT/UNMOUNT ERRORS

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error occurred.</td>
</tr>
<tr>
<td>fileerrERROR_VOLUME_CORRUPTED</td>
<td>FAT Volume corrupted.</td>
</tr>
<tr>
<td>fileerrERROR_JOURNAL_CREATION_FAILED</td>
<td>Journal file creation failed.</td>
</tr>
<tr>
<td>fileerrERROR_UNRECOGNISED_FORMAT</td>
<td>Unrecognized format.</td>
</tr>
<tr>
<td>fileerrERROR_UNSUPPORTED_SECTOR_SIZE</td>
<td>Unsupported sector size.</td>
</tr>
<tr>
<td>fileerrERROR_BOOT_SECTOR_READ_FAILED</td>
<td>Boot sector reading failed.</td>
</tr>
</tbody>
</table>

TABLE 5-3 VOLUME PRESENT STATUS

<table>
<thead>
<tr>
<th>Volume Present</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuTRUE</td>
<td>Volume is present, i.e. it was added via a xfiledevDeviceAdd() call.</td>
</tr>
<tr>
<td>cpuFALSE</td>
<td>Volume is no longer available, i.e. it was removed via a vfiledevDeviceRemove() call.</td>
</tr>
</tbody>
</table>

Return Value

None.

Note

The volume is accessible for file system operations when blsPresent = cpuTRUE.

If xMountError = 0, then all file system operations are permitted, otherwise the partition needs to be formatted.
6. Journaling

A crash caused by power failure, hardware failure or software bugs may leave the file system in an inconsistent state. The CONNECT File System overcomes this issue by implementing journaling of the file system.

A journaling file system is a file system that keeps track of the changes that will be made, by storing them in a journal (usually a circular log in a dedicated area of the file system) before committing them to the main file system. After a crash, it finds in the journal what data was being modified at the moment of the crash, and brings the file system in a consistent state.

The journaling feature can be enabled or disabled at compile time only, by setting the `filecfgjournalEnableJournaling` macro in file_cfg.h. This macro has two possible values:

<table>
<thead>
<tr>
<th>cpuENABLED</th>
<th>Enables journaling (the application size will be increased).</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpuDISABLED</td>
<td>Disables journaling.</td>
</tr>
</tbody>
</table>

6.1 How Journaling Works

If journaling is enabled, then the CONNECT File System creates a special file JOURNAL.JNL during mount operation, to log all file system write/delete operations that modify file entries and FAT entries. The CONNECT File System updates the journal file first, and then completes the user operations. After the CONNECT File System has completed the user operations successfully, it clears the data from the journal file to indicate that no user operations are pending. On the next mount, the CONNECT File System checks the journal file for pending user operations and, if any operations are found, it replays them and then clears the journal.

It is recommended that journaling is enabled for non-removable storage devices only, so the CONNECT File System has full control over the journal file and it can check and replay the file system operations that didn't complete during the last session. It is not recommended to enable journaling for removable storage devices like USB disks, SD cards, etc.

If journaling is still required for removable storage, we suggest removing JOURNAL.JNL file when the file system is modified by another OS (Windows, Linux, etc.), to avoid file system corruption when it is next time mounted with the CONNECT File System.
7. Storage Driver

The Storage Media Driver is a hardware dependent driver of the File System software stack. It provides low-level I/O functions for the rest of file system to access the storage devices. CONNECT File System provides storage drivers for accessing NAND/NOR/SD/MMC and RAM Disk devices.

The StoreDriver_t structure, as shown below, specifies the functions that the storage driver implements to interface with the file system.

All the driver functions must be implemented by the application. Each driver function must return cpuTRUE on success and cpuFALSE on failure.

```c
typedef struct xSTORE_DRIVER {
    storeInitFunctionPtrType pxInitFunction;
    storeCapacityReadFunctionPtrType pxCapacityReadFunction;
    storeReadFunctionPtrType pxReadFunction;
    storeWriteFunctionPtrType pxWriteFunction;
    storeStatusReadFunctionPtrType pxStatusReadFunction;
    storeConfigurationReadFunctionPtrType pxConfigurationReadFunction;
} StoreDriver_t;
```

**FIGURE 6-1. STORAGE I/O DRIVER STRUCTURE**

**TABLE 6-1 STORAGE I/O DEVICE DRIVER ROUTINES**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xStoreInit()</td>
<td>Initialize the storage device.</td>
</tr>
<tr>
<td>xStoreCapacityRead()</td>
<td>Read one or more blocks at a specified block address.</td>
</tr>
<tr>
<td>xStoreRead()</td>
<td>Write one or more blocks at a specified block address.</td>
</tr>
<tr>
<td>xStoreWrite()</td>
<td>Reads the status of storage device.</td>
</tr>
<tr>
<td>xStoreStatusRead()</td>
<td>Reads the configuration of storage device i.e., device properties</td>
</tr>
<tr>
<td>xStoreConfigurationRead()</td>
<td>Read one or more blocks at a specified block address.</td>
</tr>
</tbody>
</table>
7.1 Initialize Storage: xStoreInit()

This function initializes the storage device.

```c
BaseType_t xStoreInit ( StoreDevice_t *pxStoreDevice );
```

**Arguments**

pxStoreDevice  
Driver argument pointer, the usage of this pointer depends on the driver implementation.

**Return Value**

cpuTRUE if the operation is successful, cpuFALSE otherwise.
7.2 Read capacity: xStoreCapacityRead()
This function reads the storage capacity of device as number of blocks along with block size.

```c
BaseType_t xStoreCapacityRead ( StoreDevice_t *pxStoreDevice,
                                uint32_t *pulNumberOfBlocks,
                                uint32_t *pulBlockSize );
```

**Arguments**

- `pxStoreDevice` Driver argument pointer, the usage of this pointer depends on the driver implementation.
- `pulNumberOfBlocks` Pointer to store number of blocks as output
- `pulBlockSize` Pointer to store block size as output

**Return Value**

- `cpuTRUE` if the operation is successful, `cpuFALSE` otherwise.
7.3 Read data from Storage: xStoreRead()

This function reads the data from a specified blocks in the storage device.

```
BaseType_t xStoreRead ( StoreDevice_t *pxStoreDevice,
                        uint32_t ulBlockNumber,
                        uint32_t ulNumberOfBlocks,
                        void *pvBuffer,
                        uint32_t *pulReturnLength );
```

**Arguments**

- **pxStoreDevice** Driver argument pointer, the usage of this pointer depends on the driver implementation.
- **ulBlockNumber** Starting logical block address from where the data has to be read.
- **ulNumberOfBlocks** Number of logical blocks to read starting from the ulBlockNumber.
- **pvBuffer** Pointer to the buffer that stores the data read from the device.
- **pulReturnLength** Pointer to a variable that receives the number of bytes read.

**Return Value**

cpuTRUE if the operation is successful, cpuFALSE otherwise.
7.4 Write Data to Storage: xStoreWrite()

This function writes data to specified Logical unit number in storage device.

```c
BaseType_t xStoreWrite ( StoreDevice_t pxStoreDevice,
                          uint32_t ulBlockNumber,
                          uint32_t ulNumberOfBlocks,
                          void *pvBuffer,
                          uint32_t *pulReturnLength );
```

**Arguments**

- **pxStoreDevice**: Driver argument pointer, the usage of this pointer depends on the driver implementation.
- **ulBlockNumber**: Starting logical block address from where the data has to be written.
- **ulNumberOfBlocks**: Number of logical blocks to write starting from the ulBlockNumber.
- **pvBuffer**: Pointer to the buffer that has the data to be written.
- **pulReturnLength**: Pointer to a variable that receives the number of bytes written.

**Return Value**

- cpuTRUE if the operation is successful, cpuFALSE otherwise.
7.5 Read status: xStoreStatusRead()

This function is used to verify whether a logical unit is ready or not for IO operation.

```
BaseType_t xStoreStatusRead ( StoreDevice_t *pxStoreDevice,
                              uint32_t *pulStatus );
```

Arguments

pxStoreDevice Driver argument pointer, the usage of this pointer depends on the driver implementation.

pulStatus Pointer to the store status.

Return Value

cpuTRUE if Storage is ready for read/write.

cpuFALSE if Storage is busy.
7.6 Read configuration: xStoreConfigurationRead()

This function is used to read the configuration of the storage device.

```
BaseType_t xStoreConfigurationRead ( StoreDevice_t *pxStoreDevice
                                      StoreConfiguration_t *pxConfiguration );
```

**Arguments**

pxStoreDevice Driver argument pointer, the usage of this pointer depends on the driver implementation.

pxConfiguration Pointer to the store configuration

**Return Value**

cpuTRUE if Storage is ready for read/write.

cpuFALSE if Storage is busy.

### 7.6.1 Storage Configuration: StoreConfiguration_t

This structure contains vendor information and storage device properties.

```
typedef struct xSTORE_CONFIGURATION {
    uint8_t ucDeviceType; /* See Note 1 */
    uint8_t ucDeviceQualifier; /* See Note 2 */
    uint8_t ucDeviceRemovable; /* See Note 3 */
    uint8_t aucDeviceVendorId[8]; /* See Note 4 */
    uint8_t aucDeviceProductId[16]; /* See Note 5 */
    uint32_t ulProductRevisionLevel; /* See Note 6 */
    uint8_t ucMediumType; /* See Note 7 */
    BaseType_t xIsWrProtected; /* See Note 8 */
} StoreConfiguration_t
```
Note 1 Table 6-2 shows some common device types
Note 2 ucDeviceQualifier identifies the device connected to the logical unit, as shown on Table 6-3.
Note 3 If the ucDeviceRemovable parameter is set to ‘TRUE’, then the device is removable.
Note 4 Device Vendor Identification.
Note 5 Product Identification.
Note 6 Revision Level of the product.
Note 7 Table 6-4 shows the MediumType unique for each device.
Note 8 If the xIsWrProtected parameter is set to ‘TRUE’, then the device is set to be write protected.

**TABLE 6-2 DEVICE TYPES**

<table>
<thead>
<tr>
<th>Code</th>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>SBC-2</td>
<td>Direct access block device</td>
</tr>
<tr>
<td>001h</td>
<td>SSC-2</td>
<td>Sequential-access device</td>
</tr>
<tr>
<td>002h</td>
<td>SSC</td>
<td>Printer device</td>
</tr>
<tr>
<td>03h</td>
<td>SPC-2</td>
<td>Processor device</td>
</tr>
<tr>
<td>04h</td>
<td>SBC</td>
<td>Write-once device</td>
</tr>
<tr>
<td>05h</td>
<td>MMC-4</td>
<td>CD/DVD device</td>
</tr>
<tr>
<td>06h</td>
<td></td>
<td>Scanner device (obsolete)</td>
</tr>
<tr>
<td>07h</td>
<td>SBC</td>
<td>Optical memory device</td>
</tr>
<tr>
<td>08h</td>
<td>SMC-2</td>
<td>Medium changer device</td>
</tr>
<tr>
<td>09h</td>
<td></td>
<td>Communications device (obsolete)</td>
</tr>
<tr>
<td>0Ah-0Bh</td>
<td></td>
<td>Obsolete</td>
</tr>
<tr>
<td>0Ch</td>
<td>SCC-2</td>
<td>Storage array controller device</td>
</tr>
<tr>
<td>0Dh</td>
<td>SES</td>
<td>Enclosed services device</td>
</tr>
<tr>
<td>0Eh</td>
<td>RBC</td>
<td>Simplified direct-access device</td>
</tr>
<tr>
<td>0Fh</td>
<td>OCRW</td>
<td>Optical card reader/writer device</td>
</tr>
<tr>
<td>10h</td>
<td>BCC</td>
<td>Bridge controller commands</td>
</tr>
<tr>
<td>11h</td>
<td>OSD</td>
<td>Object-based Storage Device</td>
</tr>
<tr>
<td>12h</td>
<td>ADC</td>
<td>Automation/Drive Interface</td>
</tr>
<tr>
<td>13h-1Dh</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>1Eh</td>
<td></td>
<td>Well known logical unit b</td>
</tr>
<tr>
<td>1Fh</td>
<td></td>
<td>Unknown type or device</td>
</tr>
</tbody>
</table>

*a All standards are subject to revision, and parties to agreements based on this standard are encouraged to investigate the possibility of applying the most recent editions of the listed standards.

*b All well-known logical units use the same peripheral device type code

For more Details, refer to "SCSI Primary Commands-3 (SPC-3)" Revision 23, Table 83.
**TABLE 6-3 DEVICE QUALIFIER**

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000b</td>
<td>A peripheral device having the specified peripheral device type is connected to this logical unit. If the device server is unable to determine whether or not a peripheral device is connected, it also shall use this peripheral qualifier. This peripheral qualifier does not mean that the peripheral device connected to the logical unit is ready for access.</td>
</tr>
<tr>
<td>001b</td>
<td>A peripheral device having the specified peripheral device type is not connected to this logical unit. However, the device server is capable of supporting the specified peripheral device type on this logical unit.</td>
</tr>
<tr>
<td>010b</td>
<td>Reserved</td>
</tr>
<tr>
<td>011b</td>
<td>The device server is not capable of supporting a peripheral device on this logical unit. For this peripheral qualifier the peripheral device type shall be set to 1Fh. All other peripheral device type values are reserved for this peripheral qualifier.</td>
</tr>
<tr>
<td>100b-111b</td>
<td>Vendor specific</td>
</tr>
</tbody>
</table>

*For more Details refer to “SCSI Primary Commands-3 (SPC-3)” Revision 23, Table 82.*

**TABLE 6-4 MEDIUM TYPE**

<table>
<thead>
<tr>
<th>Device Type</th>
<th>Supported Command Set</th>
<th>MediumType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Access Block Device (code : 00)</td>
<td>SCSI Block Command(SBC)</td>
<td>00</td>
</tr>
</tbody>
</table>

*MediumType is unique for each device type*

*For more Details refer to “SCSI Block Commands-3 (SBC-3)” Revision 6, Section 6.3.1.*
8. Storage Device API

The storage device API allows user to add and remove storage device to/from file system.

**TABLE 8-1 STORAGE DEVICE API**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xstoreStorageDriverInit()</td>
<td>Initializes the storage driver.</td>
</tr>
<tr>
<td>xfiledevDeviceAdd()</td>
<td>Adds initialized storage device to file system.</td>
</tr>
<tr>
<td>vfiledevDeviceRemove()</td>
<td>Removes previously added storage device from file system.</td>
</tr>
</tbody>
</table>

8.1 Initialize storage driver: xstoreStorageDriverInit()

This function initializes a storage driver and prepares the storage device structure, which can be added to the file system with xfiledevDeviceAdd().

```c
BoolType_t xstoreStorageDriverInit ( char *pcStoreName,
                                     StoreDevice_t *pxStoreDevice,
                                     StoreDriver_t *pxStoreDriver,
                                     void *pvStoreDriverArgument );
```

**Arguments**

- **pcStoreName** Pointer to the name of storage device which will be used to access file system on given storage device using POSIX API.
- **pxStoreDevice** Pointer to storage device structure.
- **pxStoreDriver** Pointer to storage driver structure. It must hold all driver functions as described in Storage Driver section.
- **pvStoreDriverArgument** Storage driver argument pointer, the usage of this pointer depends on the driver implementation.

**Return Value**

On success returns cpuTRUE, otherwise returns cpuFALSE.
Example

See example provided for vfiledevDeviceRemove().

8.2 Add storage device to file system: xfiledevDeviceAdd()

This function adds initialized storage device to file system and then finds all partitions in the device and mounts the filesystem for FAT volumes if any. It calls the volume status notify callback function for all volumes (including non-FAT) if registered with finit().

```
FileError_t xfiledevDeviceAdd ( FileDevice_t *pxFileDevice,
                               StoreDevice_t *pxStoreDevice,
                               );
```

Arguments

pxFileDevice Pointer to file device structure.

pxStoreDevice Pointer to storage device structure.

Return Value

On success returns zero, otherwise returns error number.

Example

See example provided for vfiledevDeviceRemove().

8.3 Remove storage device from file system: vfiledevDeviceRemove()

This function removes a previously added storage device from file system and then unmounts all FAT volumes. It calls the volume status notify callback function for all volumes (including non-FAT) if registered with finit().

```
BoolType_t xstoreStorageDriverInit ( FileDevice_t *pxFileDevice
                                     );
```

Arguments

pxFileDevice Pointer to file device structure.

Return Value

None.
Example

```c
StoreDevice_t xStoreDevice;

StoreDriver_t xStoreDriver =
{
    /* Driver functions as described in storage driver section. */
};

FileDevice_t xFileDevice;

finit( 0 );
xstoreStorageDriverInit( "USB", &xStoreDevice, &xStoreDriver, 0 );
xfiledevDeviceAdd( &xFileDevice, &xStoreDevice );
/* …Do all file system operations... */
vfiledevDeviceRemove( &xFileDevice );
```
9. CONNECT File System Example

The following example demonstrates how to initialize the CONNECT File System and how to add a storage device.

```c
void vFileVolumeNotify( char *pcVolumePath,
                        uint8_t ucFormatType,
                        FileError_t xMountError,
                        BoolType_t bIsPresent )
{
    /* Implement this notify function as you like.
    * If xMountError is zero, then the volume is accessible for file
    * operations, otherwise you need to format it before using it. */
}

/* Global Variables */
FileDevice_t xFileDevice;
StoreDevice_t xStoreDevice;

StoreDriver_t xStoreDriver =
{
    /* see the 'Storage Driver' section to learn about prototypes,
    * and then implement all the driver functions.
    * Add all the driver functions here. */
};

/* Main application */
void main( void )
{
    FileError_t xError;
    BaseType_t xStatus;

    /* Initialize the storage device controller if any required
    * like USB host controller, SD card or NAND Flash. */

    xError = finit( vFileVolumeNotify );
    if ( fileerrERROR_NONE != xError )
    {
        printf("File System initialization failed\r\n");
        return;
    }
```
/ Add a storage device with name STORAGE, driver xStoreDriver and * driver argument as 0. File System uses xStoreDevice as output. * If the storage driver supplied here need any other structure variable, * pass it as argument to this function. See Storage Driver section * for more information about storage driver */

xStatus = xstoreStorageDriverInit( "STORAGE", 
                                      &xStoreDevice, &xStoreDriver, 0 );

if( cpuFALSE == xStatus )
{
    printf("Failed to add storage driver\n");
}

/* Add the storage device xStoreDevice, which is initialized above, * to the file system. 
* The function below should be called only when the storage device 
* is accessible */

xError = xfiledevDeviceAdd( &xFileDevice, &xStoreDevice );

if ( fileerrERROR_NONE != xError )
{
    printf("FILE initialization failed\n");
    return;
}

/* The vFileVolumeNotify() function will be called when a disk 
* partition found in the device is added, so all file operations 
* can be performed on device STORAGE as shown below. */

/* Open a file in write mode in volume 0 of device STORAGE */

fopen("STORAGE:\new.txt", "wb");

/* Open a file in read mode in volume 1 of device STORAGE */

fopen("STORAGE:1\new.txt", "rb");
### Appendix A: Error Codes

<table>
<thead>
<tr>
<th>Number</th>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>fileerrERROR_NONE</td>
<td>No error.</td>
</tr>
<tr>
<td>7001</td>
<td>fileerrERROR_WRONG_BOOT_SIGNATURE</td>
<td>Wrong boot signature.</td>
</tr>
<tr>
<td>7002</td>
<td>fileerrERROR_NO_FREE_FPOINTER</td>
<td>Free file pointer not available.</td>
</tr>
<tr>
<td>7003</td>
<td>fileerrERROR_NO_FREE_PHYSICAL_FPOINTER</td>
<td>Free physical file pointer not available.</td>
</tr>
<tr>
<td>7004</td>
<td>fileerrERROR_INVALID_PATH</td>
<td>Path is invalid.</td>
</tr>
<tr>
<td>7005</td>
<td>fileerrERROR_NO_SPACE_IN_DIRECTORY</td>
<td>No space in root directory.</td>
</tr>
<tr>
<td>7006</td>
<td>fileerrERROR_LONG_PATH</td>
<td>Path is too long.</td>
</tr>
<tr>
<td>7007</td>
<td>fileerrERROR_NAME_ALREADY_EXISTS</td>
<td>Name is already available.</td>
</tr>
<tr>
<td>7008</td>
<td>fileerrERROR_DIRECTORY_NOTEMPTY</td>
<td>Directory is not empty.</td>
</tr>
<tr>
<td>7009</td>
<td>fileerrERROR_IT_IS_NOT_DIRECTORY</td>
<td>It is not a directory.</td>
</tr>
<tr>
<td>700A</td>
<td>fileerrERROR_DIRECTORY_EMPTY</td>
<td>Directory empty.</td>
</tr>
<tr>
<td>700B</td>
<td>fileerrERROR_NO_MORE_ENTRY</td>
<td>No more entries available.</td>
</tr>
<tr>
<td>700C</td>
<td>fileerrERROR_FILE_NOT_FOUND</td>
<td>File not found.</td>
</tr>
<tr>
<td>700D</td>
<td>fileerrERROR_DISK_FULL</td>
<td>Disk is full.</td>
</tr>
<tr>
<td>700E</td>
<td>fileerrERROR_READ_ONLY_MODE</td>
<td>Read only mode.</td>
</tr>
<tr>
<td>700F</td>
<td>fileerrERROR_NO_NEXT_FOR_ROOT_DIRECTORY</td>
<td>Next entry is not available for root directory.</td>
</tr>
<tr>
<td>7010</td>
<td>fileerrERROR_NO_INFO_FOR_ROOT_DIRECTORY</td>
<td>Information not available for root directory.</td>
</tr>
<tr>
<td>7011</td>
<td>fileerrERROR_CANT_DELETE_DIRECTORY</td>
<td>Cannot delete directory.</td>
</tr>
<tr>
<td>7012</td>
<td>fileerrERROR_CRT_FAILED</td>
<td>Creation failed.</td>
</tr>
<tr>
<td>7013</td>
<td>fileerrERROR_CANT_DELETE_FILE</td>
<td>Cannot delete file.</td>
</tr>
<tr>
<td>7014</td>
<td>fileerrERROR_MUST_TYPE_NAME</td>
<td>Must type name.</td>
</tr>
<tr>
<td>Number</td>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>7015</td>
<td>fileerrERROR_INVALID_CHAR</td>
<td>Invalid character.</td>
</tr>
<tr>
<td>7016</td>
<td>fileerrERROR_IT_IS_ORPHAN</td>
<td>It is orphan.</td>
</tr>
<tr>
<td>7017</td>
<td>fileerrERROR_FAT12_UNSUPPORTED</td>
<td>FAT12 Unsupported.</td>
</tr>
<tr>
<td>7018</td>
<td>fileerrERROR_IT_IS_ROOT_DIRECTORY</td>
<td>It is root directory.</td>
</tr>
<tr>
<td>7019</td>
<td>fileerrERROR_IT_IS_NOT_A_FILE</td>
<td>It is not a file.</td>
</tr>
<tr>
<td>701A</td>
<td>fileerrERROR_TOO_LONG_NAME</td>
<td>File name is too long.</td>
</tr>
<tr>
<td>701B</td>
<td>fileerrERROR_VOLUME_POINTER_FAIL</td>
<td>Partition pointer fail.</td>
</tr>
<tr>
<td>701C</td>
<td>fileerrERROR_NO_FREE_CLUSTER</td>
<td>Free clusters are not available.</td>
</tr>
<tr>
<td>701D</td>
<td>fileerrERROR_DEVICE_IO_FAIL</td>
<td>Device I/o fail.</td>
</tr>
<tr>
<td>701E</td>
<td>fileerrERROR_MAXIMUM_DEVICE_LIMIT_REACHED</td>
<td>Max device limit reached.</td>
</tr>
<tr>
<td>701F</td>
<td>fileerrERROR_MAXIMUM_PART_LIMIT_REACHED</td>
<td>Max partitions limit reached.</td>
</tr>
<tr>
<td>7020</td>
<td>fileerrERROR_NO_VALID_VOLUME</td>
<td>No valid partition.</td>
</tr>
<tr>
<td>7021</td>
<td>fileerrERROR_BAD_VOLUME_PREFIX</td>
<td>Bad volume prefix.</td>
</tr>
<tr>
<td>7022</td>
<td>fileerrERROR_PART_ALREADY_MOUNTED</td>
<td>Partition already mounted.</td>
</tr>
<tr>
<td>7023</td>
<td>fileerrERROR_PART_NOT_MOUNTED</td>
<td>Partition not mounted.</td>
</tr>
<tr>
<td>7024</td>
<td>fileerrERROR_FAT16_UNSUPPORTED</td>
<td>FAT16 Unsupported.</td>
</tr>
<tr>
<td>7025</td>
<td>fileerrERROR_INVALID_MODE</td>
<td>Mode is invalid.</td>
</tr>
<tr>
<td>7026</td>
<td>fileerrERROR_WONLY_MODE</td>
<td>Write only mode.</td>
</tr>
<tr>
<td>7027</td>
<td>fileerrERROR_INVALIDSEEK_POSITION</td>
<td>Invalid seek position.</td>
</tr>
<tr>
<td>7028</td>
<td>fileerrERROR_CANNOTSEEK_BEFORE_STARTING_OFFSET</td>
<td>Cannot seek to before file starting offset.</td>
</tr>
<tr>
<td>7029</td>
<td>fileerrERROR_EXCEEDING_CLUSTER_CHAIN</td>
<td>Exceeding cluster chain.</td>
</tr>
<tr>
<td>702A</td>
<td>fileerrERROR_INVALID_FPOINTER</td>
<td>Invalid file pointer.</td>
</tr>
<tr>
<td>702B</td>
<td>fileerrERROR_UNFORMATED</td>
<td>Unformatted.</td>
</tr>
<tr>
<td>702C</td>
<td>fileerrERROR_UNRECOGNISED_FORMAT</td>
<td>Unrecognized format.</td>
</tr>
<tr>
<td>Number</td>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>702D</td>
<td>fileerrERROR_CAN_NOT_NAV_TO_UP</td>
<td>Cannot navigate to up.</td>
</tr>
<tr>
<td>702E</td>
<td>fileerrERROR_FAT_REGION_CORRUPTED</td>
<td>FAT corrupted.</td>
</tr>
<tr>
<td>702F</td>
<td>fileerrERROR_UNABLE_TO_FORMAT</td>
<td>Unable to format with given FAT type.</td>
</tr>
<tr>
<td>7030</td>
<td>fileerrERROR_INVALID_VOLUME_NAME</td>
<td>Invalid volume name.</td>
</tr>
<tr>
<td>7031</td>
<td>fileerrERROR_FORMAT_STR_EXCEEDED</td>
<td>Format string exceeded.</td>
</tr>
<tr>
<td>7032</td>
<td>fileerrERROR_NO_NEXT_VOLUME</td>
<td>No next volume.</td>
</tr>
<tr>
<td>7033</td>
<td>fileerrERROR_INVALID_CLUSTER_NUMBER</td>
<td>Invalid cluster number</td>
</tr>
<tr>
<td>7034</td>
<td>fileerrERROR_BASIS_NAME_FAIL</td>
<td>Basis name failed.</td>
</tr>
<tr>
<td>7035</td>
<td>fileerrERROR_CLUSTER_NOT_FOUND</td>
<td>Cluster not found.</td>
</tr>
<tr>
<td>7036</td>
<td>fileerrERROR_NO_VOLUME_SPACE</td>
<td>No volume space.</td>
</tr>
<tr>
<td>7037</td>
<td>fileerrERROR_DEVICEALLOC_FAIL</td>
<td>Device allocation failed.</td>
</tr>
<tr>
<td>7038</td>
<td>fileerrERROR_INVALID_ARGS</td>
<td>Invalid arguments.</td>
</tr>
<tr>
<td>7039</td>
<td>fileerrERROR_DEVICE_NAME_MAXIMUM_LEN</td>
<td>Device name exceeded maximum length.</td>
</tr>
<tr>
<td>703A</td>
<td>fileerrERROR_DEVICE_INIT_FAIL</td>
<td>Device initialization failed.</td>
</tr>
<tr>
<td>703B</td>
<td>fileerrERROR_DEVICE_NOT_FOUND</td>
<td>Device not found.</td>
</tr>
<tr>
<td>703C</td>
<td>fileerrERROR_DEVICE_NAME_LONG</td>
<td>Device name too long.</td>
</tr>
<tr>
<td>703D</td>
<td>fileerrERROR_NO_MORE_FILE</td>
<td>No more files.</td>
</tr>
<tr>
<td>703E</td>
<td>fileerrERROR_INVALID_PATTERN</td>
<td>Invalid pattern for finding files.</td>
</tr>
<tr>
<td>703F</td>
<td>fileerrERROR_ENTRY_UPDATE</td>
<td>Entry update failed.</td>
</tr>
<tr>
<td>7040</td>
<td>fileerrERROR_VOLUME_NOT_PRESENT</td>
<td>Volume not present.</td>
</tr>
<tr>
<td>7041</td>
<td>fileerrERRORDIRECTORY_PATH_NOT_EXISTED</td>
<td>Specified directory path not existed.</td>
</tr>
<tr>
<td>7042</td>
<td>fileerrERROR_FILE_IN_USE</td>
<td>File is in use.</td>
</tr>
<tr>
<td>7043</td>
<td>fileerrERROR_FILE_RENAME_PATH_LONG</td>
<td>File rename path is too long.</td>
</tr>
<tr>
<td>Number</td>
<td>Error Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>7044</td>
<td>fileerrERROR_UNSUPPORTED_SECTOR_SIZE</td>
<td>Unsupported sector size.</td>
</tr>
<tr>
<td>7045</td>
<td>fileerrERROR_WRONG_DESTINATION</td>
<td>Wrong destination path.</td>
</tr>
<tr>
<td>7046</td>
<td>fileerrERROR_VOLUME_INSUFFICIENT_SPACE</td>
<td>Insufficient free space in volume.</td>
</tr>
<tr>
<td>7047</td>
<td>fileerrERROR_VOLUME_CORRUPTED</td>
<td>FAT Volume corrupted.</td>
</tr>
<tr>
<td>7048</td>
<td>fileerrERROR_JOURNAL_CREATION_FAILED</td>
<td>Journal file creation failed.</td>
</tr>
<tr>
<td>7049</td>
<td>fileerrERROR_BOOT_SECTOR_READ_FAILED</td>
<td>Boot sector reading failed.</td>
</tr>
<tr>
<td>704A</td>
<td>fileerrERROR_FAT32_UNSUPPORTED</td>
<td>FAT32 Unsupported.</td>
</tr>
<tr>
<td>704B</td>
<td>fileerrERROR_DEVICE_POINTER_ALREADY_EXISTS</td>
<td>File device pointer already exists.</td>
</tr>
<tr>
<td>704C</td>
<td>fileerrERROR_DEVICE_NAME_ALREADY_EXISTS</td>
<td>Storage device name already exists.</td>
</tr>
</tbody>
</table>
high integrity systems

Americas: +1 408 625 4712
ROTW: +44 1275 395 600
Email: sales@highintegritysystems.com

Headquarters
WITTENSTEIN high integrity systems
Brown’s Court
Long Ashton Business Park
Bristol
BS41 9LB, UK

www.highintegritysystems.com