WITTENSTEIN

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

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

OPEN**RTOS** provides the only available commercial license for FreeRTOS, the highly successful, small, efficient embedded real time operating system distributed under a modified GPL license. OPEN**RTOS** and FreeRTOS share the same code base, however OPEN**RTOS** 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 OPEN**RTOS** by selecting from a wide range of middleware components and Board Support Packages.

SAFERTOS

SAFE**RTOS** 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, SAFE**RTOS** 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

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/OPEN**RTOS** for commercial products, or SAFE**RTOS**, 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, OPEN**RTOS** and SAFE**RTOS**
- 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, OPEN**RTOS** or SAFE**RTOS**.

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

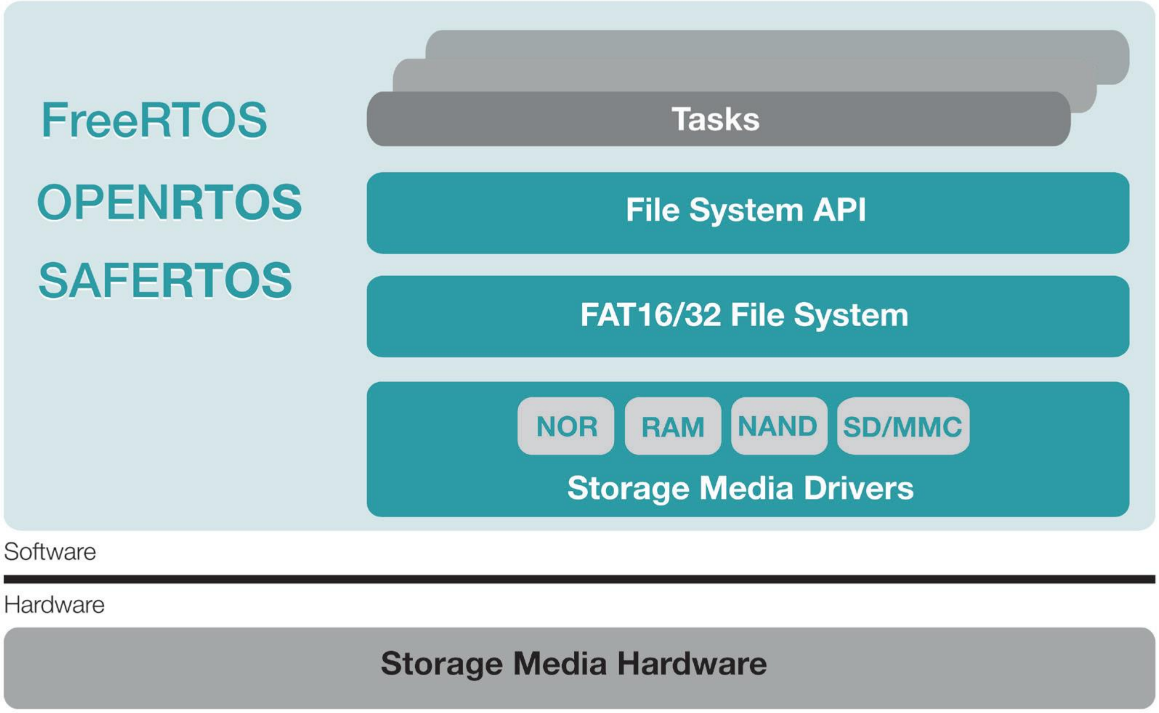

FIGURE 3-1 CONNECT FILE SYSTEM ARCHITECTURE DIAGRAM

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

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

TABLE 4-1 FILE SYSTEM POSIX API

Function	Description
ffindclose()	Close already opened path for searching.
fprintf()	Writes formatted text to file.

4.1 Open a file: fopen()

This function opens a file.

```
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

“rb”	Opens a file in binary mode for reading. The fopen call fails if the file does not exist.
“wb”	Opens an empty file in binary mode for writing. If the given file already exists, its contents are destroyed.
“ab”	Opens a file in binary mode for writing at the end of file.
“rb+”	Opens an existing file in binary mode for reading and writing.
“wb+”	Opens an empty file in binary mode for reading and writing. If the given file already exists, its contents are destroyed.
“ab+”.	Opens a file for reading and appending

Return Value

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

Example

```
FILE  *fp;
fp = fopen( “usb0:/test.txt” , ”wb” );
```

4.2 Read data from file: fread()

This function reads data from opened file.

```
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

```
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.

```
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

filecfgOPTION_ALLOC_FREE_AND_WRITE	Allocates free clusters and writes new data and then deletes old clusters. If journaling enabled, it will recover the file system if a crash happens during the write operation.
filecfgOPTION_DIRECTLY_OVER_WRITE	Directly overwrites old clusters with new data. 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.

Example

```
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.

```
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

```
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.

```
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

filefatSEEK_SET or 0	Beginning of file
filefatSEEK_CUR or 1	Current position
filefatSEEK_END or 2	End of file

Return Value

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

Example

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

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

Arguments

pxFilePointer   Pointer to FILE structure

Return Value

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

Example

```
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.

```
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

```
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.

```
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

```
int    result;

result = fformat( “usb0:/” ,  “MYVOL” , “FAT16” );
```

4.16 Delete a file: fdelete()

This function deletes a given file.

```
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

```
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.

```
int     finit     (   filecoreVolumeNotifyFunction_t  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

```
int    result;

result = finit(0);
```

4.18 Rename a File or Directory: frename()

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

```
int     frename     (  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

```
int     result;

result = frename( “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.

```
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

```
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.

```
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

```
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.

```
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

filefatDIR_DELETE_IF_EMPTY or 0	Deletes only if directory is empty.
fatfileDIR_DELETE_EVEN_NOT_EMPTY or 1	Deletes even directory has contents.

Return Value

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

Example

```
int    result;


result = frmdir( “usb0:/Dir” , filefatDIR_DELETE_IF_EMPTY );
```

4.22 Gives free space: ffree ( )

This function gives free space on specified volume.

```
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

```
unsigned int   space;



space = ffree( “usb0:/” );
```

4.23 Search files and directories: ffind ( )

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

```
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 4-6 PERMITTED FLAG VALUES

* or *.*	Search all files in the directory
abc*	Search for files that begin with “abc”, e.g. abcd.txt, abc.txt, abcx.
abc.*	Search for files with name “abc” and any extension, e.g. abc.txt, abc.jpg
*.abc	Search for files contains extension “abc”, e.g. test.abc, ram.abc
abc*.xyz	Search for files that begin with “abc” and contains extension “xyz”, e.g. abc.xyz, abcd.xyz

Return Value

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

Example

```
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.

```
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

```
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.

```
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

```
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.

```
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

```
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.

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

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.

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

TABLE 5-1 PARTITION FORMAT TYPES

0	Unrecognized format
1 or filefatFAT_12	FAT 12
2 or filefatFAT_16	FAT 16
3 or filefatFAT_32	FAT 32

TABLE 5-2 ERRORS THAT CAN OCCUR DURING DISK MOUNT/UNMOUNT ERRORS

0	No error occurred.
fileerrERROR_VOLUME_CORRUPTED	FAT Volume corrupted.
fileerrERROR_JOURNAL_CREATION_FAILED	Journal file creation failed.
fileerrERROR_UNRECOGNISED_FORMAT	Unrecognized format.
fileerrERROR_UNSUPPORTED_SECTOR_SIZE	Unsupported sector size.
fileerrERROR_BOOT_SECTOR_READ_FAILED	Boot sector reading failed.

TABLE 5-3 VOLUME PRESENT STATUS

cpuTRUE	Volume is present, i.e. it was added via a xfiledevDeviceAdd() call.
cpuFALSE	Volume is no longer available, i.e. it was removed via a vfiledevDeviceRemove() call.

Return Value

None.

Note

The volume is accessible for file system operations when bIsPresent = 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:

cpuENABLED	Enables journaling (the application size will be increased).
cpuDISABLED	Disables journaling.

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.

```
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

Function	Description
xStoreInit()	Initialize the storage device.
xStoreCapacityRead()	Read one or more blocks at a specified block address.
xStoreRead()	Write one or more blocks at a specified block address.
xStoreWrite()	Reads the status of storage device.
xStoreStatusRead()	Reads the configuration of storage device i.e., device properties
xStoreConfigurationRead()	Read one or more blocks at a specified block address.

7.1 Initialize Storage: xStoreInit()

This function initializes the storage device.

```
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.

```
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.

```
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

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

Qualifier	Description
000b	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.
001b	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.
010b	Reserved
011b	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.
100b-111b	Vendor specific
For more Details refer to "SCSI Primary Commands-3 (SPC-3)" Revision 23, Table 82.	

TABLE 6-4 MEDIUM TYPE

Device Type	Supported Command Set	MediumType
Direct Access Block Device (code : 00)	SCSI Block Command(SBC)	00
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

Function	Description
xstoreStorageDriverInit()	Initializes the storage driver.
xfiledevDeviceAdd()	Adds initialized storage device to file system.
vfiledevDeviceRemove()	Removes previously added storage device from file system.

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().

```
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

```
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.

```
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\r\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\r\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

Number	Error Code	Description
0	fileerrERROR_NONE	No error.
7001	fileerrERROR_WRONG_BOOT_SIGNATURE	Wrong boot signature.
7002	fileerrERROR_NO_FREE_FPOINTER	Free file pointer not available.
7003	fileerrERROR_NO_FREE_PHYSICAL_FPOINTER	Free physical file pointer not available.
7004	fileerrERROR_INVALID_PATH	Path is invalid.
7005	fileerrERROR_NO_SPACE_IN_DIRECTORY	No space in root directory.
7006	fileerrERROR_LONG_PATH	Path is too long.
7007	fileerrERROR_NAME_ALREADY_EXISTS	Name is already available.
7008	fileerrERROR_DIRECTORY_NOT_EMPTY	Directory is not empty.
7009	fileerrERROR_IT_IS_NOT_DIRECTORY	It is not a directory.
700A	fileerrERROR_DIRECTORY_EMPTY	Directory empty.
700B	fileerrERROR_NO_MORE_ENTRY	No more entries available.
700C	fileerrERROR_FILE_NOT_FOUND	File not found.
700D	fileerrERROR_DISK_FULL	Disk is full.
700E	fileerrERROR_READ_ONLY_MODE	Read only mode.
700F	fileerrERROR_NO_NEXT_FOR_ROOT_DIRECTORY	Next entry is not available for root directory.
7010	fileerrERROR_NO_INFO_FOR_ROOT_DIRECTORY	Information not available for root directory.
7011	fileerrERROR_CANT_DELETE_DIRECTORY	Cannot delete directory.
7012	fileerrERROR_CRT_FAILED	Creation failed.
7013	fileerrERROR_CANT_DELETE_FILE	Cannot delete file.
7014	fileerrERROR_MUST_TYPE_NAME	Must type name.

Number	Error Code	Description
7015	fileerrERROR_INVALID_CHAR	Invalid character.
7016	fileerrERROR_IT_IS_ORPHAN	It is orphan.
7017	fileerrERROR_FAT12_UNSUPPORTED	FAT12 Unsupported.
7018	fileerrERROR_IT_IS_ROOT_DIRECTORY	It is root directory.
7019	fileerrERROR_IT_IS_NOT_A_FILE	It is not a file.
701A	fileerrERROR_TOO_LONG_NAME	File name is too long.
701B	fileerrERROR_VOLUME_POINTER_FAIL	Partition pointer fail.
701C	fileerrERROR_NO_FREE_CLUSTER	Free clusters are not available.
701D	fileerrERROR_DEVICE_IO_FAIL	Device I/o fail.
701E	fileerrERROR_MAXIMUM_DEVICE_LIMIT_REACHED	Max device limit reached.
701F	fileerrERROR_MAXIMUM_PART_LIMIT_REACHED	Max partitions limit reached.
7020	fileerrERROR_NO_VALID_VOLUME	No valid partition.
7021	fileerrERROR_BAD_VOLUME_PREFIX	Bad volume prefix.
7022	fileerrERROR_PART_ALREADY_MOUNTED	Partition already mounted.
7023	fileerrERROR_PART_NOT_MOUNTED	Partition not mounted.
7024	fileerrERROR_FAT16_UNSUPPORTED	FAT16 Unsupported.
7025	fileerrERROR_INVALID_MODE	Mode is invalid.
7026	fileerrERROR_WRONLY_MODE	Write only mode.
7027	fileerrERROR_INVALID_SEEK_POSITION	Invalid seek position.
7028	fileerrERROR_CANNOT_SEEK_BEFORE_STARTING_OFFSET	Cannot seek to before file starting offset.
7029	fileerrERROR_EXCEEDING_CLUSTER_CHAIN	Exceeding cluster chain.
702A	fileerrERROR_INVALID_FPOINTER	Invalid file pointer.
702B	fileerrERROR_UNFORMATED	Unformatted.
702C	fileerrERROR_UNRECOGNISED_FORMAT	Unrecognized format.

Number	Error Code	Description
702D	fileerrERROR_CAN_NOT_NAV_TO_UP	Cannot navigate to up.
702E	fileerrERROR_FAT_REGION_CORRUPTED	FAT corrupted.
702F	fileerrERROR_UNABLE_TO_FORMAT	Unable to forma with given FAT type.
7030	fileerrERROR_INVALID_VOLUME_NAME	Invalid volume name.
7031	fileerrERROR_FORMAT_STR_EXCEEDED	Format string exceeded.
7032	fileerrERROR_NO_NEXT_VOLUME	No next volume.
7033	fileerrERROR_INVALID_CLUSTER_NUMBER	Invalid cluster number
7034	fileerrERROR_BASIS_NAME_FAIL	Basis name failed.
7035	fileerrERROR_CLUSTER_NOT_FOUND	Cluster not found.
7036	fileerrERROR_NO_VOLUME_SPACE	No volume space.
7037	fileerrERROR_DEVICE_ALLOC_FAIL	Device allocation failed.
7038	fileerrERROR_INVALID_ARGS	Invalid arguments.
7039	fileerrERROR_DEVICE_NAME_MAXIMUM_LEN	Device name exceeded maximum length.
703A	fileerrERROR_DEVICE_INIT_FAIL	Device initialization failed.
703B	fileerrERROR_DEVICE_NOT_FOUND	Device not found.
703C	fileerrERROR_DEVICE_NAME_LONG	Device name too long.
703D	fileerrERROR_NO_MORE_FILE	No more files.
703E	fileerrERROR_INVALID_PATTERN	Invalid pattern for finding files.
703F	fileerrERROR_ENTRY_UPDATE	Entry update failed.
7040	fileerrERROR_VOLUME_NOT_PRESENT	Volume not present.
7041	fileerrERROR_DIRECTORY_PATH_NOT_EXISTED	Specified directory path not existed.
7042	fileerrERROR_FILE_IN_USE	File is in use.
7043	fileerrERROR_FILE_RENAME_PATH_LONG	File rename path is too long.

Number	Error Code	Description
7044	fileerrERROR_UNSUPPORTED_SECTOR_SIZE	Unsupported sector size.
7045	fileerrERROR_WRONG_DESTINATION	Wrong destination path.
7046	fileerrERROR_VOLUME_INSUFFICIENT_SPACE	Insufficient free space in volume.
7047	fileerrERROR_VOLUME_CORRUPTED	FAT Volume corrupted.
7048	fileerrERROR_JOURNAL_CREATION_FAILED	Journal file creation failed.
7049	fileerrERROR_BOOT_SECTOR_READ_FAILED	Boot sector reading failed.
704A	fileerrERROR_FAT32_UNSUPPORTED	FAT32 Unsupported.
704B	fileerrERROR_DEVICE_POINTER_ALREADY_EXISTS	File device pointer already exists.
704C	fileerrERROR_DEVICE_NAME_ALREADY_EXISTS	Storage device name already exists.

WITTENSTEIN

highintegritysystems

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