NVTFAT File System Library

API Specification V1.2

Mar 9, 2017
Released

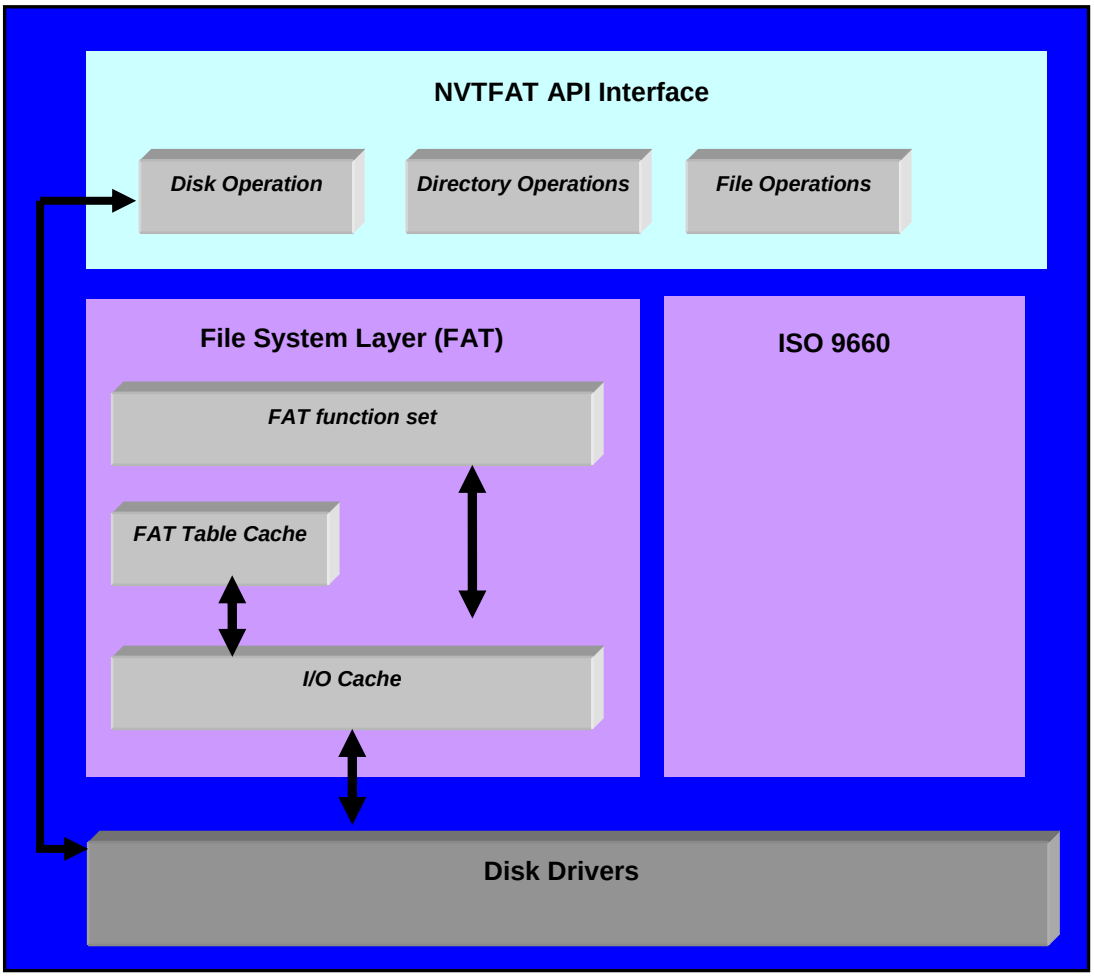
1. General Description

1.1 Features

The NVTFAT File System Library has the following features:

- Support FAT12/FAT16/FAT32
- Support multiple disks and multiple partitions
- Dynamically mount and unmount disk
- Support sub-directory
- Support long file name. The length of file name can be up to 514 characters. The length of file path, including the file name, can be up to 520 characters.
- Can format flash memory cards
- Get disk physical size and free space
- Can open at most 32 files at the same time
- Open files with create, truncate, append
- Create, delete, rename, move, copy, seek, read, and write files
- Enumerate files under a directory
- Get file position and get file status
- Set file size and set file attributes
- Create, rename, remove, and move directories

2. Programming Guide



2.1 Initialize File System

To initialize this file system, just invoke *fsInitFileSystem()*. The underlying disk driver should be initialized followed the file system initialization.

2.2 Error Codes

Because the file operation may fail due to various reasons, it's strongly recommended that application should check the return value of each file system API call. The File System Library provides very detailed error code to indicate the error reasons.

2.3 File Handle

File handle is a handle obtained by opening a file. Application should check the return value of *fsOpenFile()*. If the return value > 0, it's a valid handle. Otherwise, some error happened in the file opening operation. A file handle is valid until the file was closed by *fsCloseFile()*.

2.4 Format Flash Memory Card

The File System Library provides *fsFormatFlashMemoryCard()* to format flash memory card, such as SD, MMC, CF, or Smart Media. This function requires caller to pass a physical disk pointer as parameter, which can be obtained by *fsGetFullDiskInfomation()*.

The formatting of File System Library was fully compliant to Smart Media disk format standard. The rules of disk formatting are defined in table 2-1.

Table 2-1 Disk Format

Disk Size	FAT Type	Cluster Size	Capacity
1 MB	FAT12	4 KB	984 KB
2 MB	FAT12	4 KB	1984 KB
4 MB	FAT12	8 KB	3976 KB
8 MB	FAT12	8 KB	7976 KB
16 MB	FAT12	16 KB	15968 KB
32 MB	FAT12	16 KB	31968 KB
64 MB	FAT12	16 KB	63952 KB
128 MB	FAT16	16 KB	127936 KB
256 MB	FAT16	32 KB	255744 KB
512 MB	FAT16	32 KB	511744 KB
1024 MB	FAT16	32 KB	1023616 KB
2048 MB	FAT16	32 KB	2047288 KB

2.5 File Operations

Many of the file operations can be done only if the file has been opened. These file operations determine the target by file handle. In this section, all file operations based on file handle will be introduced.

2.5.1 Open File

To read or write a file, applications must first open the file and obtain a file handle, which is an integer. Function *fs0penFile()* is used to open a file. If the opening file operation succeed, the caller will obtain a file handle, whose value is >= 3000. Otherwise, the call will receive a negative value, which represented an error code (refer to Error Code Table).

Function *fs0penFile()* receives two parameters. The first parameter is the full path file name of the file to be opened. Both long file name or short file name are acceptable and are non-case-sensitive. The full path file name must also include disk number. For example, "C:\\OpenATestFile.txt" or "C:\\OpenAT~1.txt". The second parameter is combination of control flags. It use bit-OR to represent various control flags. The control flags and their effectives are listed in Table 2-2.

Table 2-2 File opening control flags

Flag	Description
O_RDONLY	Open with read capability. In addition, O_DIR and O_APPEND have implicit read capability.
O_WRONLY	Open with write capability. In addition, O_APPEND, O_CREATE, and O_TRUNC have implicit write capability.
O_RDWR	Open with read and write capabilities
O_APPEND	Open an exist file and set the file access position to end of file. O_APPEND has implicit read and write capabilities.
O_CREATE	Open or create a file. If the file did not exist, File System Library would create it. Otherwise, if the file existed, File System Library would just open it and set file access position to start of file. O_CREATE has implicit write capability.
O_TRUNC	Open an existed file and truncate it. If the file did not exist, return an error code. If the file existed, open it. O_TRUNC has implicit write capability.
O_FSEEK	File system will create cluster chain for this file to speed up file seeking operation. It will allocate 1KB extra memory.

2.5.2 File Access Position

Each opened file has one and only one access position. Subsequent *fsReadFile()* and *fsWriteFile()* operations are started from the file access position. File access position can be obtained by *fsGetFilePosition()* and can be changed by *fsFileSeek()*.

When a file was opened, the file access position was initially set as 0, that is, start of file. The only exception is a file opened with O_APPEND flag. In this case, the file access position will be set as end of file.

When file access position is at the end of file, *fsReadFile()* will result in EOF error, while *fsWriteFile()* will extend the file size.

2.5.3 Read File

A file cannot be read unless it was opened. *fsReadFile()* was used to read data from a file. It receives a file handle as the first parameter, which was previously obtained by *fsOpenFile()*. The general scenario of reading files is:

fsOpenFile() → fsReadFile() → fsCloseFile()

2.5.4 Write File

A file cannot be written unless it was opened with write capability. *fsWriteFile()* was used to write data to a file. It receives a file handle as the first parameter, which was previously obtained by *fsOpenFile()*. The general scenario of writing files is:

fsOpenFile() → fsWriteFile() → fsCloseFile()

2.6 Directory Operations

File System Library supports sub-directory and provides supporting routines to manage directories. It supports directory creation, remove, rename, and move.

2.6.1 Create/Remove Directories

fsMakeDirectory() can be used to create a new directory. Directory name can be long file name, and the name must not be conflict with any existed files or sub-directories under the same directory.

fsRemoveDirectory() can be used to remove an empty directory. If there's any files or sub-directories under the directory to be removed, an error will be received. Root directory cannot be removed.

2.6.2 Move/Rename Directories

A directory can be completely moved from a directory to another directory. *fsMoveFile()* can be used to move directory. All files and sub-directories under that directory will be completely moved at the same time. If the target directory contained a file or directory whose name was conflicted with the directory to be moved, the operation will be canceled.

A directory can be renamed with *fsRenameFile()*. If the new name will be conflicted with any existed files or directories under the same directory, the operation will be canceled.

2.6.3 Delete/Rename/Move Files

A file can be deleted with *fsDeleteFile()*. All disk space occupied by this file will be released immediately and can be used by other files.

A file can be moved from a directory to another directory with *fsMoveFile()*. If the target directory contained a file or directory whose name was conflicted with the file to be moved, the operation will be canceled.

A file can be renamed with *fsRenameFile()*. If the name will be conflicted with any existed files or directories under the same directory, the operation will be canceled.

2.6.4 Enumerate Files In a Directory

File System Library provides a set of functions to support enumerating files under a specific directory. These functions are *fsFindFirst()*, *fsFindNext()*, and *fsFindClose()*.

First, user uses *fsFindFirst()* to specify the directory to be searched, and specify search conditions. If there's any files or sub-directories match the search conditions, *fsFindFirst()* will return 0 and user can obtain a file-find object (FILE_FIND_T) . The file-find object contains information of the first found file, including file name and attributes. User can use the same file-find object to do the subsequent searches by calling *fsFindNext()*. Each call to *fsFindNext()* will obtain a newly found file or sub-directory, if it returns 0. *fsFindNext()* returns non-zero value means that there's no any other files or sub-directories match the search conditions and the file enumeration should be terminated. User should call *fsFindClose()* to terminate a search series.

3. File System Library API

3.1 Disk Operations

❖ *fsDiskFreeSpace*

SYNOPSIS	INT fsDiskFreeSpace(INT nDriveNo, UINT32 *puBlockSize, UINT32 *puFreeSize, INT32 *puDiskSize)
DESCRIPTION	Get free space of disk <driveNo>
PARAMETER	< nDriveNo > - drive number, for example 'C'
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`UINT32 uBlockSize, uFreeSize, uDiskSize;`
	`...`
	`if (fsDiskFreeSpace ('C', &blockSize, &freeSize,`
	`&diskSize) == FS_OK)`
	`printf("Disk C block size=%d, free space=%d MB,`
	`disk size=%d MB\n", blockSize, (INT)freeSize/1024,`
	`(INT)diskSize/1024);`

❖ *fsFormatFlashMemoryCard*

SYNOPSIS	INT fsFormatFlashMemoryCard(PDISK_T *ptPDisk)
DESCRIPTION	Format a flash memory card by FAT12/FAT16/FAT32 format. NVTFAT will first create a MBR for this disk and configure it as single partition. Then NVTFAT will format it as FAT12/FAT16 format.
PARAMETER	<ptPDisk> - The pointer refer to the physical disk descriptor
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`PDISK_T *ptPDiskList, *ptPDisk;`
	`PARTITION_T *ptPartition;`
	
	`/* Get complete disk information */`
	`ptPDiskList = fsGetFullDiskInfomation();`
	
	`/* Format the first physical disk */`
	`ptPartition = ptPDiskList;`
	`ptPDisk = ptPDiskList; /* format the first physical disk */`
	`fsFormatDiskPartition(ptPDisk);`
	
	`/* Release allocated memory */`
	`FS_ReleaseDiskInformation(pDiskList);`

❖ *fsTwoPartAndFormatAll*

SYNOPSIS	INT fsTwoPartAndFormatAll(PDISK_T *ptPDisk, INT firstPartSize, INT secondPartSize)
DESCRIPTION	Configure the disk as two partitions and format these two partitions as FAT32 format. If the total size of these two partitions are larger than disk size, NVTFAT will automatically shrink the size of the second partition to fit disk size.
PARAMETER	<ptPDisk> - The pointer refer to the physical disk descriptor < firstPartSize> - The size (in KBs) of the first partition. < secondPartSize> - The size (in KBs) of the second partition.
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`PDISK_T *ptPDiskList, *ptPDisk;`
	`PARTITION_T *ptPartition;`
	
	`/* Get complete disk information */`
	`ptPDiskList = fsGetFullDiskInfomation();`
	
	`/* Format the first physical disk */`
	`ptPartition = ptPDiskList;`
	`ptPDisk = ptPDiskList; /* format the first physical disk */`
	`fsTwoPartAndFormatAll(ptPDisk, 2048, 10240);`
	
	`/* Release allocated memory */`
	`FS_ReleaseDiskInformation(pDiskList);`

❖ *fsAssignDriveNumber*

SYNOPSIS	INT fsAssignDriveNumber (INT nDriveNo, INT disk_type, INT instance, INT partition)
DESCRIPTION	Claim the drive number assignment.
PARAMETER	<nDriveNo> - The drive number. Valid number is 'A' ~ 'Z'. <disk_type> - Disk type define in nvtfat.h. Prefixed with "DISK_TYPE_". For example, NAND disk type is DISK_TYPE_SMART_MEDIA. <instance> - The disk instance of specified <disk_type>, start from 0. For example, the first NAND disk is instance 0, the second NAND is instance 1. <partition> - Which partition of the specified <disk_type><instance>. The first partition is 1, the second partition is 2, and so on.
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`// SD0 first partition => C`
	`fsAssignDriveNumber('C', DISK_TYPE_SD_MMC, 0, 1);`
	
	`// NAND0 first partition => E`
	`fsAssignDriveNumber('E', DISK_TYPE_SMART_MEDIA, 0, 1);`
	
	`// NAND1 first partition => H`
	`fsAssignDriveNumber('H', DISK_TYPE_SMART_MEDIA, 1, 1);`
	
	`// NAND1 second partition => I`
	`fsAssignDriveNumber('I', DISK_TYPE_SMART_MEDIA, 1, 2);`

❖ *fsFormatFixedDrive*

SYNOPSIS	INT fsFormatFixedDrive (INT nDriveNo)
DESCRIPTION	Format the specified drive. The drive number must be have been successfully assigned by fsAssignDriveNumber().
PARAMETER	<nDriveNo> - The drive number. Valid number is 'A' ~ 'Z'.
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`#define DISK_TYPE_SMART_MEDIA 0x00000008 // defined in nvtfat.h`
	`#define DISK_TYPE_SD_MMC 0x00000020 // defined in nvtfat.h`
	
	`fsAssignDriveNumber('C', DISK_TYPE_SD_MMC, 0, 1);`
	`fsAssignDriveNumber('E', DISK_TYPE_SMART_MEDIA, 0, 1);`
	
	`fsFormatFixedDrive('C');`
	`fsFormatFixedDrive('E');`

❖ *fsGetFullDiskInfomation*

SYNOPSIS	PDISK_T *fsGetFullDiskInfomation(VOID)
DESCRIPTION	Get the complete information list of physical disk, disk partitions, and logical disk information. The returned PDISK_T pointer was referred to a dynamically allocated memory, which contains the complete disk information list. Note that caller is responsible to deallocate it by calling fsReleaseDiskInformation().
PARAMETER	None
RETURN VALUE	NULL – There's no disk mounted or no available memory. Otherwise – A PDISK_T pointer referred to the complete disk information.

EXAMPLE	`PDISK_T *pDiskList, *ptPDiskPtr;`
	`PARTITION_T *ptPartition;`
	`INT nDiskIdx = 0;`
	`INT nPartIdx;`
	`ptPDiskPtr = pDiskList = fsGetFullDiskInfomation();`
	`while (ptPDiskPtr != NULL)`
	`{`
	`printf("\n\n=== Disk %d (%s) ==========\n", nDiskIdx++, (ptPDiskPtr->nDiskType & DISK_TYPE_USB_DEVICE) ? "USB" : "IDE");`
	`printf(" name: [%s%s]\n", ptPDiskPtr->szManufacture, ptPDiskPtr->szProduct);`
	`printf(" head: [%d]\n", ptPDiskPtr->nHeadNum);`
	`printf(" sector: [%d]\n", ptPDiskPtr->nSectorNum);`
	`printf(" cylinder: [%d]\n", ptPDiskPtr->nCylinderNum);`
	`printf(" size: [%d MB]\n", ptPDiskPtr->uDiskSize / 1024);`
	`ptPartition = ptPDiskPtr->ptPartList;`

```
    nPartIdx = 1;
    while (ptPartition != NULL)
    {
        printf("\n    --- Partition %d --------------------\n",
               nPartIdx++);
        printf("        active: [%s]\n",
              (ptPartition->ucState & 0x80) ? "Yes" : "No");
        printf("        size:   [%d MB]\n",
              (ptPartition->uTotalSecN / 1024) / 2);
        printf("     start: [%d]\n", ptPartition->uStartSecN);
        printf("        type:   ");
        ptPartition = ptPartition->ptNextPart;
     }
     ptPDiskPtr = ptPDiskPtr->ptPDiskAllLink;
}
fsReleaseDiskInformation(pDiskList);
FS_ReleaseDiskInformation(pDiskList);
```

❖ *fsReleaseDiskInformation*

SYNOPSIS	VOID fsReleaseDiskInformation(PDISK_T *ptPDiskList)
DESCRIPTION	Release the memory allocated by fsGetFullDiskInfomation()
PARAMETER	<ptPDiskList> - The PDISK_T pointer returned by the previous call to fsGetFullDiskInfomation()
RETURN VALUE	None
EXAMPLE	See example code of fsGetFullDiskInfomation()

❖ *fsInitFileSystem*

SYNOPSIS	VOID fsInitFileSystem(VOID)
DESCRIPTION	Initialize file system
PARAMETER	None
RETURN VALUE	None

EXAMPLE	`sysEnableCache(CACHE_WRITE_THROUGH);`
	`fsInitFileSystem();`
	`fmiInitDevice();`
	`fmiInitSDDevice();`

❖ *fsFixDriveNumber*

SYNOPSIS	INT fsFixDriveNumber(CHAR sd_drive, CHAR sm_drive, CHAR cf_drive)
DESCRIPTION	Specify the fixed driver number of SD card, SM/NAND, and CF. If the specified drive number was used, NVTFAT will find other driver number for it. This API must be called prior to fsInitFileSystem().
PARAMETER	sd_drive - 'A' ~ 'Z' sm_drive - 'A' ~ 'Z' cf_drive - 'A' ~ 'Z'
RETURN VALUE	0 - Success ERR_DRIVE_INVALID_NUMBER - invalid drive number

EXAMPLE	`fsFixDriveNumber('D', 'C', 'F');`
	
	`fsInitFileSystem();`
	
	`...`

❖ *fsPhysicalDiskConnected*

SYNOPSIS	INT fsPhysicalDiskConnected(PDISK_T *ptPDisk)
DESCRIPTION	Register and parsing a newly detected disk.
PARAMETER	<ptPDisk> - The pointer refer to the physical disk descriptor
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table
EXAMPLE	None.

3.2 File/Directory Operations

❖ *fsCloseFile*

SYNOPSIS	INT fsCloseFile(INT hFile)
DESCRIPTION	Close a file, that was previously opened by fsOpenFile()
PARAMETER	<hFile> - The file handle of the file to be closed
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table
EXAMPLE	`Refer to the example of fsOpenFile`

❖ *fsDeleteFile*

SYNOPSIS	INT fsDeleteFile(CHAR *suFileName, CHAR *szAsciiName)
DESCRIPTION	Delete a file
PARAMETER	<suFileName> - The unicode full path file name of the file to be opened. The file name must include its absolute full path with drive number specified. The full path file name must be ended with two 0x00 character. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. This parameter is optional. Caller must set this parameter as NULL if it was not used. If caller did not give the ASCII name, NVTFAT will generate the ASCII version name from the <suFileName>. Note that if two-bytes code language was used in <suFileName>, NVTFAT generated ASCII version name will be incorrect. It was suggested to set this parameter if two-bytes code language contained in <suFileName>.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR suFileName[] = { 'C', 0, ':', 0, '\\', 0, 'l', 0, 'o', 0, 'g', 0, '.', 0, 't', 0, 'x', 0, 't', 0, 0, 0};`
	`INT nStatus;`
	`/* Delete file C:\log.txt */`
	`nStatus = fsDeleteFile(suFileName, NULL);`
	`if (nStatus < 0)`

	`printf("Cannot delete file log.txt!\n");`

❖ *fsFileSeek*

SYNOPSIS	INT64 fsFileSeek(INT64 hFile, INT n64Offset, INT16 usWhence)
DESCRIPTION	Set the current read/write position of an opened file
PARAMETER	<hFile> - The file handle of the opened file to be set position <n64Offset> - Byte offset from the position indicated by <usWhence> <usWhence> - Seek position base

usWhence	New position
SEEK_SET	“file offset 0” + <nOffset>
SEEK_CUR	“file current position” + <nOffset>
SEEK_END	“end of file position”+ <nOffset>

RETURN VALUE	< 0 – error code defined in Error Code Table Otherwise – New read/write position of this file

EXAMPLE	`INT hFile, nReadLen;`
	`CHAR suFileName[] = { ‘C’, 0, ‘:’, 0, ‘\\’, 0, ‘l’, 0, ‘o’, 0,`
	`‘g’, 0, ‘.’, 0, ‘t’, 0, ‘x’, 0, ‘t’, 0, 0, 0};`
	`UINT8 pucBuff[64];`
	
	`if ((hFile = fsOpenFile(suFileName, NULL, O_RDONLY) < 0)`
	`return hFile;`
	`/* read 10 bytes from file offset 1000 */`
	`fsFileSeek(hFile, 1000, SEEK_SET);`
	`fsReadFile(hFile, pucBuff, 10, &nReadLen)`
	`fsCloseFile(hFile);`

❖ *fsFindClose*

SYNOPSIS	INT fsFindClose(FILE_FIND_T *ptFindObj)
DESCRIPTION	Close a search series
PARAMETER	<ptFindObj> - The file-search object obtained by previous fsFindFirst() call
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table
EXAMPLE	`Refer to the example of fsFindFirst()`

❖ *fsFindFirst*

SYNOPSIS	INT fsFindFirst (CHAR *suDirName, CHAR *szAsciiName, FILE_FIND_T *ptFindObj)
DESCRIPTION	Start a file search and get the first file/directory entry found
PARAMETER	<suDirName> - The unicode full path name of the directory to be searched. The name must include its absolute full path with drive number specified. The full path name must be ended with two 0x00 characters. <szAsciiName> - The ASCII version name of <suDirName> excluding the path part. This parameter is optional. Caller must set this parameter as NULL if it was not used. If caller did not give the ASCII name, NVTFAT will generate the ASCII version name from the <suDirName>. Note that if two-bytes code language was used in <suDirName>, NVTFAT generated ASCII version name will be incorrect. It was suggested to set this parameter if two-bytes code language contained in <suDirName>. < ptFindObj > - caller prepared file/directory entry container
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`INT ListDir(CHAR *szPath)`
	`{`
	`  INT  nIdx, nStatus;`
	`  CHAR  szMainName[12], szExtName[8], *pcPtr;`

```
    FILE_FIND_T   tFileInfo;
    memset((UINT8*)&tFileInfo, 0, sizeof(tFileInfo));
    nStatus = fsFindFirst(szPath, NULL, &tFileInfo);
    if (nStatus < 0)
        return nStatus;
    do
    {
        pcPtr = tFileInfo.szShortName;
        if ((tFileInfo.ucAttrib & A_DIR) &&
           (!strcmp(pcPtr, ".") || !strcmp(pcPtr, "..")))
            strcat(tFileInfo.szShortName, ".");
         memset(szMainName, 0x20, 9);
         szMainName[8] = 0;
         memset(szExtName, 0x20, 4);
         szExtName[3] = 0;
         i = 0;
         while (*pcPtr && (*pcPtr != '.'))
              szMainName[i++] = *pcPtr++;
         if (*pcPtr++)
         {
             nIdx = 0;
             while (*pcPtr)
                szExtName[nIdx++] = *pcPtr++;
         }
        if (tFileInfo.ucAttrib & A_DIR)
           printf("%s %s    <DIR> %02d-%02d-%04d %02d:%02d %s\n",
           szMainName, szExtName, tFileInfo.ucWDateMonth,
```

```
            tFileInfo.ucWDateDay, tFileInfo.ucWDateYear+80)%100 ,
            tFileInfo.ucWTimeHour, tFileInfo.ucWTimeMin,
            tFileInfo.szLongName);
         else
            printf("%s %s %10d  %02d-%02d-%04d  %02d:%02d  %s\n",
            szMainName, szExtName, (UINT32)tFileInfo.nFileSize,
            tFileInfo.ucWDateMonth, tFileInfo.ucWDateDay,
           (tFileInfo.ucWDateYear+80)%100, tFileInfo.ucWTimeHour,
            tFileInfo.ucWTimeMin, tFileInfo.szLongName);
      } while (!fsFindNext(&tFileInfo));
      fsFindClose(&tFileInfo);
}
```

❖ *fsFindNext*

SYNOPSIS	INT fsFindNext(FILE_FIND_T *ptFindObj)
DESCRIPTION	Continue the previous fsFindFirst() file search and get the next matched file. If there's no more match found, the search series will be closed automatically.
PARAMETER	<ptFindObj> - the file-search object used in the previous fsFindFirst() call
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table
EXAMPLE	`Refer to the example of fsFindFirst()`

❖ *fsGetFilePosition*

SYNOPSIS	INT fsGetFilePosition(INT hFile, UINT32 *puPos)
DESCRIPTION	Get the current read/write position of an opened file
PARAMETER	<hFile> - The file handle of the opened file to get file read/write position <puPos> - The current read/write position
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`INT hFile, nStatus;`
	`UINT32 uFilePos;`
	`/* Open a read-only file */`
	`hFile = fsOpenFile(file, O_RDONLY);`
	`fsFileSeek(hFile, 1000, SEEK_SET);`
	`fsGetFilePosition(hFile, &uFilePos);`
	`printf("Current file position is: %d\n", uFilePos);`
	`fsCloseFile(hFile);`

❖ *fsGetFileSize*

SYNOPSIS	INT fsGetFileSize(INT hFile)
DESCRIPTION	Get the current size of an opened file
PARAMETER	<hFile> - The file handle of the opened file to get size
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`INT hFile, nStatus;`
	`UINT32 uFilePos;`
	`/* Open a read-only file */`
	`hFile = fsOpenFile(file, O_RDONLY);`
	`sysPrintf("The size of %s is %d\n", file, fsGetFileSize(hFile));`
	`fsCloseFile(hFile);`

❖ *fsGetFileStatus*

SYNOPSIS	INT fsGetFileStatus(INT hFile, CHAR *suFileName, CHAR *szAsciiName, FILE_STAT_T *ptFileStat)
DESCRIPTION	Get file status of a specific file or directory
PARAMETER	<hFile> - The file handle of the opened file to be gotten status <suFileName> - The unicode full path file name of the file to be opened. It was used only if <hFile> is < 0. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. <ptFileStat> - Caller prepared container to receive status of this file
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR suFileName[] = { 'C', 0, ':', 0, '\\', 0, 'l', 0, 'o', 0, 'g', 0, '.', 0, 't', 0, 'x', 0, 't', 0, 0, 0};`
	`FILE_STAT_T tFileStat;`
	`INT nStatus`
	`if ((nStatus = fsGetFileStatus(-1, suFileName, NULL , &stat)) < 0)`
	`{`
	`  printf("fsGetFileStatus failed\n");`
	`  fsGetErrorDescription(nStatus, NULL, 1);`
	`  return nStatus;`
	`}`

❖ *fsMakeDirectory*

SYNOPSIS	INT fsMakeDirectory(CHAR *suDirName, CHAR *szAsciiName)
DESCRIPTION	Create a new directory, if not exists.
PARAMETER	<suDirName> - The unicode full path name of the directory to be created. < szAsciiName > - The ASCII version name of <suDirName> excluding the path part.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR suDirName[] = { 'C', 0, ':', 0, '\\', 0, 't', 0, 'e', 0, 'm', 0, 'p', 0, 0, 0};`
	`/* Create a new directory "temp" under "C:\" */`
	`fsMakeDirectory(suDirName, NULL);`

❖ *fsMoveFile*

SYNOPSIS	INT fsMoveFile(CHAR *suOldName, CHAR *szOldAsciiName, CHAR *suNewName, CHAR *szNewAsciiName, INT bIsDirectory)
DESCRIPTION	Move a file or a whole directory
PARAMETER	< suOldName > - The unicode full path name of the file/directory to be moved. < szOldAsciiName > - The ASCII version name of < suOldName > excluding the path part. < suNewName > - The unicode full path name of the old file/directory to be moved to. < szNewAsciiName > - The ASCII version name of < suNewName > excluding the path part. < bIsDirectory > - TRUE: is moving a directory; FALSE: is moving a file
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR szOldFile[] = "C:\log.txt"`
	`CHAR szNewFile[] = "C:\temp\log,txt"`
	`CHAR suOldFile[128], suNewFile[128];`
	
	`fsAsciiToUnicode(szOldFile, suOldFile, TRUE);`
	`fsAsciiToUnicode(szNewFile, suNewFile, TRUE);`
	`fsMoveFile(suOldFile, NULL, suNewFile, "log.txt", FALSE);`

❖ *fsCopyFile*

SYNOPSIS	INT fsCopyFile(CHAR *suSrcName, CHAR *szSrcAsciiName, CHAR *suDstName, CHAR *szDstAsciiName)
DESCRIPTION	Copy a file. (Copy directory was not allowed.)
PARAMETER	< suSrcName > - The unicode full path name of the file to copied. < szSrcAsciiName > - The ASCII version name of < suSrcName > excluding the path part. < suDstName > - The unicode full path name of the file/directory to be generated. < szDstAsciiName > - The ASCII version name of < suDsrName > excluding the path part.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table
EXAMPLE	

❖ *fsOpenFile*

SYNOPSIS	INT fsOpenFile(CHAR *suFileName, CHAR *szAsciiName,UINT32 uFlag)
DESCRIPTION	Open/Create a file
PARAMETER	<suFileName> - The unicode full path file name of the file to be opened. The file name must include its absolute full path with drive number specified. The full path file name must be ended with two 0x00 character. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. This parameter is optional. Caller must set this parameter as NULL if it was not used. If caller did not give the ASCII name, NVTFAT will generate the ASCII version name from the <suFileName>. Note that if two-bytes code language was used in <suFileName>, NVTFAT generated ASCII version name will be incorrect. It was suggested to set this parameter if two-bytes code language contained in <suFileName>. <uFlag> - O_RDONLY – open file with read capability O_WRONLY – open file with write capability O_RDWR – open file with both read and write capabilities O_APPEND – open file with write-append operation, the file position was set to end of file on open O_CREATE – If the file exists, open it. If the file is not exists, create it. O_TRUC - Open a file and truncate it, file size becomes 0.

	O_DIR - open a directory file
RETURN VALUE	< 0 – error code defined in Error Code Table Otherwise - file handle

EXAMPLE	`INT hFile;`
	`CHAR suFileName[] = { 'C', 0, ':', 0, '\\', 0, 'l', 0, 'o', 0,`
	`'g', 0, '.', 0, 't', 0, 'x', 0, 't', 0, 0, 0};`
	`CHAR szAsciiName[] = "log.txt";`
	`/* Open a read-only file */`
	`hFile = fsOpenFile(suFileName, szAsciiName, O_RDONLY);`
	`if (hFile < 0)`
	`return hFile;`
	`fsCloseFile(hFile);`

❖ *fsReadFile*

SYNOPSIS	INT fsReadFile(INT hFile, UINT8 *pucBuff, INT nBytes, INT *pnReadCnt)
DESCRIPTION	Read <nBytes> of octets from an opened file
PARAMETER	<hFile> - The file handle of an opened file <pucBuff> - Refer to the buffer to receive data read from the specified file <pnReadCnt> - Number of bytes to read <pnReadCnt> - Number of bytes actually read
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	UINT8 pucBuff[4096];
	INT hFileSrc, hFileOut;
	INT nReadLen, nWriteLen, nStatus;
	if ((hFileSrc = fsOpenFile("C:\\log.txt", O_RDONLY) < 0)
	return hFileSrc;
	if ((hFileOut = fsOpenFile("C:\\logcopy.txt", O_CREATE) < 0)
	return hFileOut;
	while (1)
	{
	if ((nStatus = fsReadFile(hFileSrc, pucBuff, 4096,
	&nReadLen) < 0)
	break;
	if ((nStatus = fsWriteFile(hFileOut, pucBuff, nReadLen,
	&nWriteLen);
	break;
	if ((nReadLen < 4096) || (nWriteLen != nReadLen)

```
            break;
        }
        fsCloseFile(hFileSrc);
        fsCloseFile(hFileOut);
```

❖ *fsRemoveDirectory*

SYNOPSIS	INT fsRemoveDirectory(CHAR *suDirName, CHAR *szAsciiName)
DESCRIPTION	Remove an empty directory. If the directory is not empty, an ERR_DIR_REMOVE_NOT_EMPTY error will be returned.
PARAMETER	<suDirName> - The unicode full path name of the directory to be removed. < szAsciiName > - The ASCII version name of <suDirName> excluding the path part.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR szDirName[] = "C:\temp"`
	`CHAR suDirName[128];`
	`fsAsciiToUnicode(szDirName, suDirName, TRUE);`
	`/* Remove directory "C:\temp" */`
	`nStatus = fsRemoveDirectory(suDirName, "temp");`

❖ *fsRenameFile*

SYNOPSIS	INT fsRenameFile(CHAR *suOldName, CHAR *szOldAsciiName, CHAR *suNewName, CHAR *szNewAsciiName, BOOL bIsDirectory)
DESCRIPTION	Rename a file or directory
PARAMETER	< suOldName > - The unicode full path name of the file/directory to be renamed. < szOldAsciiName > - The ASCII version name of < suOldName > excluding the path part. < suNewName > - The unicode full path name of the old file/directory to be renamed as. < szNewAsciiName > - The ASCII version name of < suNewName > excluding the path part. < bIsDirectory > - TRUE: is renaming a directory; FALSE: is renaming a file
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR szOldFile[] = "C:\log.txt"`
	`CHAR szNewFile[] = "C:\log2.txt"`
	`CHAR suOldFile[128], suNewFile[128];`
	
	`fsAsciiToUnicode(szOldFile, suOldFile, TRUE);`
	`fsAsciiToUnicode(szNewFile, suNewFile, TRUE);`
	`fsRenameFile(suOldFile, NULL, suNewFile, "log2.txt", FALSE);`

❖ *fsSetFileAttribute*

SYNOPSIS	INT fsSetFileAttribute(INT hFile, CHAR *suFileName, CHAR *szAsciiName, UINT8 ucAttrib, FILE_STAT_T *ptFileStat);
DESCRIPTION	Modify file attribute of a specific file or directory.
PARAMETER	<hFile> - The file handle of the opened file to be set attribute, if the file has been opened. <suFileName> - The unicode full path file name of the file to be set attribute. It was used only if <hFile> is < 0. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. <ptFileStat> - The specified file attribute
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR szFileName[] = "C:\temp"`
	`CHAR suFileName[128];`
	`FILE_STAT_T tFileStat;`
	`fsGetFileStatus(-1, suFileName, NULL, &fFileStat)`
	`/* force changing file to be hidden */`
	`tFileStat.ucAttrib |= FA_HIDDEN;`
	`fsSetFileAttribute(-1, suFileName, NULL, &tFileStat);`

❖ *fsSetFileSize*

SYNOPSIS	INT fsSetFileSize(INT hFile, CHAR *suFileName, CHAR *szAsciiName, UINT32 nNewSize)
DESCRIPTION	Resize a file. If specified new size is larger than the current size, NVTFAT will allocate disk space and extend this file. On the other hand, if specified new size is smaller than the current size, this file will be truncated.
PARAMETER	<hFile> - The file handle of the opened file to be resized <suFileName> - The unicode full path file name of the file to be set size. It was used only if <hFile> is < 0. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. <newSize> - New file size to be extended to or truncated as
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`int ChangeFileSize(INT hFile, INT32 uLen)`
	`{`
	`  if (fsSetFileSize(hFile, NULL, NULL, uLen) < 0)`
	`  printf("fsSetFileSize error!!\n");`
	`}`

❖ *fsSetFileTime*

SYNOPSIS	INT fsSetFileTime(INT hFile, CHAR *suFileName, CHAR *szAsciiName, UINT8 ucYear, UINT8 ucMonth, UINT8 ucDay, UINT8 ucHour, UINT8 ucMin, UINT8 ucSec);
DESCRIPTION	Set the date/time attribute of a file/directory. Note that fsSetFileTime() will set the last access date and modify date/time, but the create date/time was left unchanged.
PARAMETER	<hFile> - The file handle of the opened file to be set date/time <suFileName> - The unicode full path file name of the file to be set time. It was used only if <hFile> is < 0. <szAsciiName> - The ASCII version name of <suFileName> excluding the file path. <ucYear> - Years from 1980. For example, for 2003, <year> is equal to 23. <ucMonth> - 1 <= month <= 12 <ucHour> - 0 <= hour <= 23 <ucMin> - 0 <= min <= 59 <unSec> - 0 <= sec <= 59
RETURN VALUE	0 – Success Otherwise – error code defined in Error Code Table
EXAMPLE	

❖ *fsWriteFile*

SYNOPSIS	INT fsWriteFile(INT hFile, UINT8 *pucBuff, INT nBytes, INT *pnWriteCnt)
DESCRIPTION	Write <nBytes> of octets to an opened file
PARAMETER	<hFile> - The file handle of an opened file <pucBuff> - The buffer contains the data to be written <nBytes> - Number of bytes to written <pnWriteCnt> - Number of bytes actually written
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	int CopyFile(int hFileSrc, int hFileOut)
	{
	UINT8 pucBuff[4096];
	INT nReadLen, nWriteLen, nStatus;
	while (1)
	{
	if (fsReadFile(hFileSrc, pucBuff, 4096, &nReadLen) < 0)
	break;
	fsWriteFile(hFileOut, pucBuff, nReadLen, &nWriteLen);
	if ((nReadLen < 4096) || (nWriteLen != nReadLen)
	break;
	}
	}

3.3 Language Support

❖ *fsUnicodeToAscii*

SYNOPSIS	INT fsUnicodeToAscii(VOID *pvUniStr, VOID *pvASCII, BOOL bIsNullTerm)
DESCRIPTION	Translate an Unicode string into an ASCII string. This function can only translate single byte language (for example, English). If the unicode string contained two-bytes code language (for example, BIG5 or GB), the translation result will be wrong, because NVTFAT has no built-in Unicode-ASCII translation table.
PARAMETER	<pvUniStr> - The unicode string to be translated. It must be ended with two 0x0 characters. <pvASCII> - Caller prepared container to accommodate the translation result < bIsNullTerm> - Add a NULL character (0x0) to the end of pvASCII
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR suRoot[] = { 'C', 0, ':', 0, '\\', 0, 0, 0 };`
	`CHAR szLongName[MAX_FILE_NAME_LEN/2];`
	`FILE_FIND_T tFileInfo;`
	`fsFindFirst(suRoot, NULL, &tFileInfo); /* C:\ */`
	`do`
	`{`
	`fsUnicodeToAscii(tFileInfo.suLongName, szLongName, TRUE);`
	`printf("%s\n szLongName);`
	`} while (!fsFindNext(&tFileInfo));`
	`fsFindClose(&tFileInfo);`

❖ *fsAsciiToUnicode*

SYNOPSIS	INT fsAsciiToUnicode(VOID *pvASCII, VOID *pvUniStr, BOOL bIsNullTerm)
DESCRIPTION	Translate an ASCII string into an Unicode string. This function can only translate single byte language (for example, English). If the ASCII string contained two-bytes code language (for example, BIG5 or GB), the translation result will be wrong, because NVTFAT has no built-in ASCII-Unicode translation table.
PARAMETER	<pvASCII> - The ASCII string to be translated. It must be NULL-terminated. <pvUniStr> - Caller prepared container to accommodate the translation result < bIsNullTerm> - Add two 0x0 characters to the end of pvUnicode
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`CHAR szDirName[] = "C:\temp"`
	
	`CHAR suDirName[128];`
	
	`fsAsciiToUnicode(szDirName, suDirName, TRUE);`
	
	`/* Remove directory "C:\temp" */`
	
	`nStatus = fsRemoveDirectory(suDirName, "temp");`

❖ *fsUnicodeNonCaseCompare*

SYNOPSIS	INT fsUnicodeNonCaseCompare(VOID *pvUnicode1, VOID *pvUnicode2)
DESCRIPTION	Compare two Unicode strings by case non-sensitive. The Unicode strings must be ended with two 0x0 characters.
PARAMETER	<pvUnicode1> - The source (0x0,0x0)-ended Unicode string to compared. < pvUnicode2> - The target (0x0,0x0)-ended Unicode string to compared.
RETURN VALUE	0 - The two Unicode strings are treated as equal Otherwise - The two Unicode strings are treated as non-equal

EXAMPLE	`CHAR szName1[] = "log.txt"`
	`CHAR szName2[] = "Log.TXT";`
	`CHAR suName1[32], suName2[32];`
	`fsAsciiToUnicode(szName1, suName1, TRUE);`
	`fsAsciiToUnicode(szName2, suName2, TRUE);`
	`if (fsUnicodeNonCaseCompare(suName1, suName2) == 0)`
	`  sysPrintf("Equal!\n");`
	`else`
	`  sysPrintf("Non-equal!");`

❖ *fsUnicodeCopyStr*

SYNOPSIS	INT fsUnicodeCopyStr(VOID *pvStr1, VOID *pvStr2)
DESCRIPTION	Copy an Unicode string
PARAMETER	<pvStr1> - The Unicode string to be copied to <pvStr2> - The source Unicode string. It must be (0x0,0x0)-ended.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table

EXAMPLE	`FILE_FIND_T tFileInfo;`
	`CHAR suSlash[] = { '\\', 0x00, 0x00, 0x00 };`
	`CHAR suFullName[MAX_PATH_LEN];`
	`INT nLen, nStatus;`
	`fsFindFirst(suDirName, NULL, &tFileInfo);`
	`do`
	`{`
	`  fsUnicodeCopyStr(suFullName, suDirName);`
	`  fsUnicodeStrCat(suFullName, suSlash);`
	`  fsUnicodeStrCat(suFullName, tFileInfo.suLongName);`
	`  fsDeleteFile(suFullName, NULL);`
	`} while (!fsFindNext(&tFileInfo));`
	`fsFindClose(&tFileInfo);`

❖ *fsUnicodeStrCat*

SYNOPSIS	INT fsUnicodeStrCat(VOID *pvUniStr1, VOID *pvUniStr2)
DESCRIPTION	Concate two (0x0,0x0)-ended Unicode strings
PARAMETER	< pvUniStr1> - The Unicode string to be concated to < pvUniStr2> - The Unicode to be concated to the end of < pvUniStr1>.
RETURN VALUE	FS_OK – Success Otherwise – error code defined in Error Code Table
EXAMPLE	`Refer to the example of fsUnicodeCopyStr()`

4. Error Code Table

CODE NAME	Value	Description
ERR_FILE_EOF	0xFFFF8200	end of file
ERR_GENERAL_FILE_ERROR	0xFFFF8202	general file error
ERR_NO_FREE_MEMORY	0xFFFF8204	no available memory
ERR_NO_FREE_BUFFER	0xFFFF8206	no available sector buffer
ERR_NOT_SUPPORTED	0xFFFF8208	operation was not supported
ERR_UNKNOWN_OP_CODE	0xFFFF820A	unrecognized operation code
ERR_INTERNAL_ERROR	0xFFFF820C	file system internal error
ERR_FILE_NOT_FOUND	0xFFFF8220	file not found
ERR_FILE_INVALID_NAME	0xFFFF8222	invalid file name
ERR_FILE_INVLAID_HANDLE	0xFFFF8224	invalid file handle
ERR_FILE_IS_DIRECTORY	0xFFFF8226	the file to be opened is a directory
ERR_FILE_IS_NOT_DIRECTORY	0xFFFF8228	the directory to be opened is a file
ERR_FILE_CREATE_NEW	0xFFFF822A	can not create new directory entry
ERR_FILE_OPEN_MAX_LIMIT	0xFFFF822C	number of opened files has reached limitation
ERR_FILE_RENAME_EXIST	0xFFFF822E	rename file conflict with an existent file
ERR_FILE_INVALID_OP	0xFFFF8230	invalid file operation
ERR_FILE_INVALID_ATTR	0xFFFF8232	invalid file attribute
ERR_FILE_INVALID_TIME	0xFFFF8234	invalid time specified
ERR_FILE_TRUNC_UNDER	0xFFFF8236	truncate file underflow, size < pos
ERR_FILE_NO_MORE	0xFFFF8238	Actually not an error, used to identify end of file in the enumeration of a directory
ERR_FILE_IS_CORRUPT	0xFFFF823A	file is corrupt

ERR_PATH_INVALID	0xFFFF8260	invalid path name
ERR_PATH_TOO_LONG	0xFFFF8262	path too long
ERR_PATH_NOT_FOUND	0xFFFF8264	path not found
ERR_DRIVE_NOT_FOUND	0xFFFF8270	drive not found, the disk may have been unmounted
ERR_DRIVE_INVALID_NUMBER	0xFFFF8272	invalid drive number
ERR_DRIVE_NO_FREE_SLOT	0xFFFF8274	Can not mount more drive
ERR_DIR_BUILD_EXIST	0xFFFF8290	Try to build an existent directory
ERR_DIR_REMOVE_MISS	0xFFFF8292	Try to remove a nonexistent directory
ERR_DIR_REMOVE_ROOT	0xFFFF8294	try to remove root directory
ERR_DIR_REMOVE_NOT_EMPTY	0xFFFF8296	try to remove a non-empty directory
ERR_DIR_DIFFERENT_DRIVE	0xFFFF8298	specified files on different drive
ERR_DIR_ROOT_FULL	0xFFFF829A	FAT12/FAT16 root directory full
ERR_DIR_SET_SIZE	0xFFFF829C	try to set file size of a directory
ERR_READ_VIOLATE	0xFFFF82C0	user has no read privilege
ERR_WRITE_VIOLATE	0xFFFF82C2	user has no write privilege
ERR_ACCESS_VIOLATE	0xFFFF82C4	can not access
ERR_READ_ONLY	0xFFFF82C6	try to write a read-only file
ERR_WRITE_CAP	0xFFFF82C8	try to write file/directory which was opened with read-only
ERR_NO_DISK_MOUNT	0xFFFF8300	there's no any disk mounted
ERR_DISK_CHANGE_DIRTY	0xFFFF8302	disk change, buffer is dirty
ERR_DISK_REMOVED	0xFFFF8304	portable disk has been removed
ERR_DISK_WRITE_PROTECT	0xFFFF8306	disk is write-protected
ERR_DISK_FULL	0xFFFF8308	disk full

ERR_DISK_BAD_PARTITION	0xFFFF830A	bad partition
ERR_DISK_UNKNOWN_PARTITION	0xFFFF830C	unknown or not supported partition type
ERR_DISK_UNFORMAT	0xFFFF830E	disk partition was not formatted
ERR_DISK_UNKNOWN_FORMAT	0xFFFF8310	unknown disk format
ERR_DISK_BAD_BPB	0xFFFF8312	bad BPB, disk may not be formatted
ERR_DISK_IO	0xFFFF8314	disk I/O failure
ERR_DISK_IO_TIMEOUT	0xFFFF8316	disk I/O time-out
ERR_DISK_FAT_BAD_CLUS	0xFFFF8318	bad cluster number in FAT table
ERR_DISK_IO_BUSY	0xFFFF831A	I/O device is busy writing, must retry. direct-write mode only
ERR_DISK_INVALID_PARM	0xFFFF831C	invalid parameter
ERR_DISK_CANNOT_LOCK	0xFFFF831E	cannot lock disk, the disk was in-use or locked by other one
ERR_SEEK_SET_EXCEED	0xFFFF8350	file seek set exceed end-of-file
ERR_ACCESS_SEEK_WRITE	0xFFFF8352	try to seek a file which was opened for written
ERR_FILE_SYSTEM_NOT_INIT	0xFFFF83A0	file system was not initialized
ERR_ILLEGAL_ATTR_CHANGE	0xFFFF83A2	illegal file attribute change