# Freescale MQX™ MFS™ – User Guide

MQXMFSUG
Rev. 2.7
02/2014

freescale™

***How to Reach Us:***

**Home Page:**
freescale.com

**Web Support:**
freescale.com/support



Document Number: MQXMFSUG
Rev. 2.7, 02/2014

freescale™

## Chapter 1
## Before You Begin



## Chapter 2
## Using MFS



## Chapter 3
## Reference: Functions



## Chapter 4
## Reference: Data Types

# Revision History

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to freescale.com and navigate to Design Resources>Software and Tools>All Software and Tools>Freescale MQX Software Solutions.

The following revision history table summarizes changes contained in this document.

| Revision Number | Revision Date | Description of Changes |
|---|---|---|
| Rev. 0 | 01/2009 | Initial Release coming with MQX 3.0 |
| Rev. 0B | 04/2009 | Text edited and formatting changed for MQX 3.1 release. |
| Rev. 2 | 01/2010 | Updated for MQX 3.5. New configuration options described (MFSCFG_READ_ONLY_CHECK and similar). |
| Rev. 2.1 | 03/2010 | Example for IO_IOCTL_GET_DATE_TIME corrected. |
| Rev. 2.2 | 08/2010 | Chapter 3.8.1.9 IO_IOCTL_FIND_NEXT_FILE updated. |
| Rev. 2.3 | 07/2011 | Chapter 3.8.1.15 IO_IOCTL_FREE_SPACE updated. |
| Rev. 2.4 | 08/2012 | Partition Manager Device Driver - related parts of the Chapter 3 updated. |
| Rev. 2.5 | 11/2012 | Minor changes in _io_part_mgr_install section. |
| Rev. 2.6 | 06/2013 | Language improvements and editing for MQX 4.0.2 Beta1 release. |
| Rev. 2.7 | 10/2013 | Updated content to reflect the switch from MQX types to C99 types. |

# Chapter 1 Before You Begin

## 1.1 About This Book

This book is a guide and a reference manual for using the MQX™ MFS™ Embedded File System which is a part of Freescale MQX Real-Time Operating System distribution.

This *MFS™ User's Guide* is written for experienced software developers who have a working knowledge of the C language and the target processor.

## 1.2 Where to Look for More Information

- Release Notes, accompanying Freescale MQX release, provide information that was not available at the time this User Guide was published.
- The *MQX User Guide* describes how to create embedded applications that use MQX RTOS.
- The *MQX Reference Manual* describes prototypes for the MQX API.

## 1.3 Typographic Conventions

Throughout this book, we use typographic conventions to distinguish terms.

| Font style | Usage | Example |
|---|---|---|
| **Bold** | Function families | The **_io_mfs** family of functions. |
| **Bold** | Function names | **_io_mfs_install()** |
| *Italic* | Data types (simple) | *uint32_t* |
| | Data types (complex) | See following example. |
| Constant-width | Code and code fragments | — |
| | Data types in prototype definitions | See following example. |
| | Directives | #include "*mfs.h*" |
| | Code and code fragments | |
| *Italic* | Filenames and path names | *part_mgr.h* |
| *Italic* | Symbolic parameters that you substitute with your values. | See following example. |
| *UPPERCASE Italic* | Symbolic constants | *MFS_NO_ERROR* |

### 1.3.1 Example: Prototype Definition, Including Symbolic Parameters

```
uint32_t _io_mfs_install(
  FILE_PTR  dev_fd,
```

```
char *  identifier,
uint32_t    partition_num)
```

### 1.3.2 Example: Complex Data Types and their Field Names

The structure *MFS_DATE_TIME_PARAM* contains the following fields:

- *DATE_PTR*
- *TIME_PTR*

## 1.4 Other Conventions

### 1.4.1 Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

| | |
|---|---|
| **CAUTION** | If an application calls read and write functions with the partition manager, the file system will be corrupted. |

# Chapter 2 Using MFS

## 2.1 MFS at a Glance

MFS provides a library of functions that is compatible with the Microsoft MS-DOS file system. The functions let an embedded application access the file system in a manner that is compatible with MS-DOS Interrupt 21 functions. All the functions guarantee that the application task has a mutually exclusive access to the file system.

MFS is a device driver that an application must install over a lower-level device driver. Examples of lower-level drivers are drivers for memory devices, flash disks, floppy disks, or partition-manager devices. MFS uses the lower-level driver to access the hardware device.

MFS functions do the following:

- Traverse MS-DOS directory structure.
- Create and remove subdirectories.
- Find files.
- Create and delete files.
- Open and close files.
- Read from files and write to files.
- View and modify file characteristics.
- Get the amount of free space in the file system.

## 2.2 MS-DOS File System Characteristics

### 2.2.1 Directory Path Names

MFS allows an application to traverse a directory tree. When you specify a directory path, you can use \ and / as directory separators.

You can specify a directory path in one of two ways:

- By starting with a directory separator — the path is assumed to be an absolute path.
- By starting without a directory separator — the path is assumed to be relative to the current directory.

### 2.2.2 File Attributes

Each file entry in the MS-DOS file system has an attribute byte associated with it. The attribute byte is described in more detail in the following table.

#### 2.2.2.1 Bit Number

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Meaning if bit is set to one. |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | x | Read-only file |
| | | | | | | x | | Hidden file |
| | | | | | x | | | System file |
| | | | | x | | | | Volume label |
| | | | x | | | | | Directory name |
| | | x | | | | | | Archived file |
| x | x | | | | | | | RESERVED |

The volume-label and directory-name bits are mutually exclusive.

#### 2.2.2.2 Volume Label

A file entry can be marked as a volume label. There can be only one volume label in a file system and it must reside in the root directory. Also, that label cannot act as a directory name.

### 2.2.3 File Time

Each file entry has a 16-bit time field associated with it. In MFS, the time is written into the field when the file entry is created, when the file is closed, and as a result of calling *IO_IOCTL_SET_DATE_TIME*. The format of the time field is as follows:

| Element | Bits used | Values |
|---|---|---|
| Seconds | 0 – 4 | 0 – 29<br>(multiply by two for seconds) |
| Minutes | 5 – 10 | 0 – 60 |
| Hours | 11 – 15 | 0 – 24<br>(24-hour clock) |

### 2.2.4 File Date

Each file entry has a 16-bit date field associated with it. In MFS, the date is written into the field when the file entry is created, when the file is closed, and as a result of calling *IO_IOCTL_SET_DATE_TIME*. The format of the date field is as follows:

| Element | Bits used | Values |
|---|---|---|
| Days | 0 – 4 | 1 – 31 |
| Months | 5 – 8 | 1 – 12 |
| Year | 9 – 15 | 0 – 119<br>(1980 – 2099) |

### 2.2.5 File Allocation Table

The MS-DOS file system can have multiple copies of the file allocation table. MFS updates as many FATs as it is configured for. It only reads from the first FAT.

### 2.2.6 Filename Wildcards

The characters `*` and `?` are treated as wildcards in a filename.

## 2.3 High-Level Formatting

An application can perform high-level formatting on a disk by calling **ioctl()**. The function writes a new boot sector, deallocates all clusters in the file allocation table, and deletes all entries in the root directory.

There is one input/output control command that formats the disk, and one that formats and checks for bad clusters.

The *MFS_IOCTL_FORMAT_PARAM* structure is used:

```
typedef struct mfs_ioctl_format
{
   MFS_FORMAT_DATA_PTR     FORMAT_PTR;
   uint32_t *                    COUNT_PTR;  /* To count bad clusters */
} MFS_IOCTL_FORMAT_PARAM, * MFS_IOCTL_FORMAT_PARAM_PTR;
```

The first variable is a pointer to a *MFS_FORMAT_DATA* structure described below. The second is **uint32_t *** that points to the **uint32_t** variable which is used to contain the count of bad sectors. It is used only if the *IO_IOCTL_FORMAT_TEST* function is used.

```
typedef struct mfs_format_data
{
   unsigned char                                    PHYSICAL_DRIVE;
   unsigned char                                   MEDIA_DESCRIPTOR;
   uint16_t                                        BYTES_PER_SECTOR;
   uint16_t                                       SECTORS_PER_TRACK;
   uint16_t                                         NUMBER_OF_HEADS;
   uint32_t                                       NUMBER_OF_SECTORS;
   uint32_t                                          HIDDEN_SECTORS;
   uint16_t                                        RESERVED_SECTORS;
} MFS_FORMAT_DATA,                               * MFS_FORMAT_DATA_PTR;
```

The *MFS_FORMAT_DATA* structure has the following fields:

- *PHYSICAL_DRIVE*
  - 0x00 for floppy disks; 0x80 for hard disks.
- *MEDIA_DESCRIPTOR*
  - 0xFD for 5.25" 360 K diskettes.
  - 0xF9 for 5.25" 1200 K diskettes.
  - 0xF9 for 3.5" 720 K diskettes.
  - 0xF0 for 3.5" 1440 K diskettes and other removable media.
  - 0xF8 for hard disk and other non-removable media.

- *BYTES_PER_SECTOR*
  - Size of a block in bytes (usually 512).
- *SECTORS_PER_TRACK*
  - Number of sectors in a track.
- *NUMBER_OF_HEADS*
  - Number of disk heads.
- *NUMBER_OF_SECTORS*
  - Total number of sectors on the disk including reserved sectors.
- *HIDDEN_SECTORS*
  - For hard disks, it is the number of sectors from the beginning of the disk to the beginning of the partition. This is the same number as the *relative sectors* field in a hard disk partition table. For floppy disks, the field is zero.
- *RESERVED_SECTORS*
  - Number of sectors from the beginning of the file system to the first FAT sector. It is usually one.

## 2.4 Version of MFS

The constant *MFS_VERSION* defines the version and revision numbers for MFS.

## 2.5 Customizing MFS

The following constant definitions can be overridden to customize MFS. To override any of these definitions, simply define the desired value in the */config/<board>/user_config.h* file.

`#define MFSCFG_MINIMUM_FOOTPRINT 1`
- Normally not defined. Define to build MFS for small memory devices.

`#define MFSCFG_READ_ONLY 0`
- Set to one to build MFS in read-only mode without create, write, or format capability. This reduces the code size and may be useful in certain applications such as boot loaders. Set to one to enable write functionality.

`#define MFSCFG_READ_ONLY_CHECK 1`
- Enable runtime checking whether or not an underlaying device operates in read-only mode. This check is made before any attempt to write to a device and returns gracefully with an appropriate error code. The read-only mode is detected by using the IO_IOCTL_DEVICE_IDENTIFY command once when opening MFS device. The mode detected is used until the device is closed.

`#define MFSCFG_READ_ONLY_CHECK_ALLWAYS 0`
- When this option is set and `MFSCFG_READ_ONLY_CHECK` is also set, the read-only mode is checked at the device level always by using the IO_IOCTL_DEVICE_IDENTIFY command and not only during an open call. Use this option with devices where read-only state may change in run-time (e.g. SD Card storage).

`#define MFSCFG_ENABLE_FORMAT 1`
- Set to one to build MFS with the format command, zero otherwise.

`#define MFSCFG_CALCULATE_FREE_SPACE_ON_OPEN 1`

— Set to one to calculate the available free space on the drive when the drive is mounted. Calculating the available free space is time-consuming on large drives, as the entire FAT must be read. When set to zero, this operation is deferred until the first time the free space is required which may be never.

```
#define MFSCFG_MINIMUM_FOOTPRINT 1
```

— Set to one to build MFS for small-memory devices, zero otherwise.

```
#define MFSCFG_MAX_READ_RETRIES 1
#define MFSCFG_MAX_WRITE_RETRIES 1
```

— Number of times MFS attempts to read or write to the device unsuccessfully before it reports an error.

```
#define MFSCFG_FAT_CACHE_SIZE 2
```

— Maximum number of sectors of the file allocation table that MFS keeps stored in memory. On systems with a lot of memory, increasing this number such that the entire FAT fits in memory will increase the speed of MFS because it performs fewer read and write operations on the disk.

```
#define MFSCFG_NUM_OF_FATS 2
```

— This parameter is only used when formatting and specifies the number of file allocation tables that will be placed on the drive. A minimum of one is required. The first FAT is the one used by MFS, and the others are backups. Microsoft Windows uses two as its standard. If you choose one, MFS operates somewhat faster when it writes to the disk because it has half the number of FAT write operations to do.

```
#define MFSCFG_HANDLE_INITIAL 4
#define MFSCFG_HANDLE_GROW 4
#define MFSCFG_HANDLE_MAX 0
```

— Number of initial file handles, the amount to raise the file handles by, when none are available, and the maximum number of simultaneous file handles.

```
#define MFSCFG_FIND_TEMP_TRIALS 300
```

— Number of times MFS will attempt to create a unique temporary filename before returning an error.

Sector boundaries determine the cluster size, the root directory size, and the FAT type that is used when formatting a disk. These can be modified in *mfs.h* to customize the formatting of the disk. By default, the following values are used:

| | # Sectors | FAT Type | Root directory entries | Cluster size in sectors |
|---|---|---|---|---|
| *SECTOR_BOUND1* | 2048 | FAT12 | 7 × 32 | 1 |
| *SECTOR_BOUND2* | 4096 | FAT12 | 14 × 32 | 1 |
| *SECTOR_BOUND3* | 8192 | FAT12 | 32 × 32 | 2 |
| *SECTOR_BOUND4* | 16384 | FAT12 | 32 × 32 | 4 |
| *SECTOR_BOUND5* | 32768 | FAT12 | 32 × 32 | 2 |
| *SECTOR_BOUND6* | 524288 | FAT16 | 32 × 32 | 8 |
| *SECTOR_BOUND7* | 1048576 | FAT16 | 32 × 32 | 16 |
| *SECTOR_BOUND7a* | 2097152 | FAT16 | 32 × 32 | 32 |
| *SECTOR_BOUND8* | 16777216 | FAT32 | 64 k | 8 |

| | # Sectors | FAT Type | Root directory entries | Cluster size in sectors |
|---|---|---|---|---|
| *SECTOR_BOUND9* | 33554432 | FAT32 | 64 k | 16 |
| *SECTOR_BOUND10* | 67108864 | FAT32 | 64 k | 32 |

**Table 2-1. Summary: MFS Functions**

| | |
|---|---|
| **_io_mfs_install** | Installs MFS. |
| **_io_mfs_uninstall** | Uninstalls MFS. |
| **fclose** | Closes the file or device. |
| **fopen** | Opens the file or device. |
| **ioctl** | Issues a control command. |

Functions are described in “Reference: Functions”

## 2.6 Partition Manager Device Driver

The partition manager device driver is designed to be installed under the MFS device driver. It lets the MFS work independently of the multiple partitions on a disk. It also enforces mutually exclusive access to the disk which means that two concurrent write operations from two different MFS devices cannot be in conflict. The partition manager device driver can remove partitions as well as create new ones.

The partition manager device driver creates multiple primary partitions. It does not support extended partitions.

The partition manager device driver is installed and opened like other devices. It must also be closed and uninstalled when an application no longer needs it.

An application follows these steps to use the partition manager. Functions are described in Chapter 3, “Reference: Functions.”

1. Installs the partition manager (**_io_part_mgr_install()**).
2. Opens the partition manager (**fopen()**).
3. Issues input/output control commands (**ioctl()**).
4. Closes the partition manager (**fclose()**).
5. Uninstalls the partition manager device driver (**_io_part_mgr_uninstall()**).

| | |
|---|---|
| **CAUTION** | If an application calls read and write functions with the partition manager, the file system will be corrupted. |

**Table 2-2. Summary: Partition Manager Device Driver Functions**

| | |
|---|---|
| **_io_part_mgr_install** | Installs the partition manager device driver. |
| **_io_part_mgr_uninstall** | Uninstalls the partition manager device driver. |

**Table 2-2. Summary: Partition Manager Device Driver Functions**

| | |
|---|---|
| **fclose** | Closes the partition manager. |
| **fopen** | Opens the partition manager. |
| **ioctl** | Issues a control command to the partition manager. |

## 2.7 Working with Removable Media

Removable-media devices are a class of device, in which the medium, upon which files are written to and read from, can be inserted and removed. Examples include:

- USB mass storage devices (flast drives, and so on)
- ATA PCMCIA (PC card) flash cards
- SD Cards
- removable hard drives
- floppy-disk drives

An application that installs MFS on the removable media must take some standard precautions.

### 2.7.1 Buffering and Caching

MFS maintains three internal buffers or caches:

- The FAT cache, two or more sectors in size.
- A directory-sector buffer, exactly one sector in size.
- A file-sector buffer, exactly one sector in size.

When writing, an application can control how the buffers are flushed. There are three modes:

- *WRITE_THROUGH* — the buffer contents are immediatly written to disk when modified.
- *WRITE_BACK* — the buffer contents are written to disk on application command, or prior to reading a different sector.
- *MIXED_MODE* — the buffer contents are written to disk on application command, prior to reading a different sector, or when a file is closed.

If MFS detects that the lower-layer device is removable, the FAT cache will be placed in write through mode, and the directory and file caches will be placed in mixed mode. If the lower-layer device is not removable, all caches will be placed in write back mode.

An application can modify the cache modes with the appropriate **ioctl()** calls. When using removable media, the application must ensure that all files are closed and the MFS device itself is closed before the media is removed. These steps ensure that the caches are flushed and the media is updated.

### 2.7.2 Writing to Media

Writing to the media, either to partition the media, format the media, or write a file, must be completed before the media is removed. If the media is removed during a write operation, the media may be corrupted.

### 2.7.3 Hotswapping

With MFS, an application can implement hotswapping. To properly implement hotswapping, however, the lower-layer device must support a mechanism for notifying the application that the media is removed or inserted.

When an application detects that the media has been inserted, it must do the following:

1. Open the lower-layer device.
2. Optionally install the partition manager on the device.
3. If the partition manager is installed, open the partition manager.
4. Install MFS on the device or on the partition manager if the partition manager is installed.
5. Open the MFS device.

When an application detects that the media has been removed, it must do the following:

1. Close all files that are open on the device.
2. Close the MFS device.
3. Uninstall the MFS device.
4. If the partition manager is installed, close it.
5. If the partition manager is installed, uninstall it.
6. Close the lower-layer device.

#### 2.7.3.1 Example: Hotswapping

For an example that demonstrates hotswapping with a USB flash drive, see: *mfs\example\mfs_usb*.

# Chapter 3 Reference: Functions

## 3.1 In This Chapter

Alphabetically sorted prototype definitions for MFS and the partition manager device driver.

## 3.2 _io_mfs_install

Install MFS.

### Synopsis

```
uint32_t _io_mfs_install(
     /*[IN] the device on which to install MFS */
     FILE_PTR     dev_fd,
     /*[IN] Name to be given to MFS (e.g. "C:", "MFS1:") */
     /* The name must end in a colon ":" */
     char *     identifier,
     /*[IN] Partition number to install MFS on. */
     /* 0 for no partitions  */
     uint32_t      partition_num)
```

### Description

The function initializes MFS and allocates memory for all of the internal MFS data structures. It also reads some required drive information from the disk, on which it is installed. MFS supports FAT12, FAT16, and FAT32 file systems. If the disk has a different file system or if it is unformatted, you can use MFS to format it to one of the supported file systems.

If the application uses a partitioned disk, you must install MFS on a partition manager device driver. The partition manager device driver can create partitions on the disk if there are none. It can also remove partitions.

Usage of *partition_num* parameter is deprecated - *_io_mfs_install* should obtain handle to partition manager associated with particular partition as *dev_fd*. *partition_num* parameter should be set to 0 which instructs MFS to simply use the *dev_fd* as underlying device.

### Return Codes

Returns a **uint32_t** error code.

- *IO_EOF*
  - The **FILE_PTR** passed into **_io_mfs_install()** was NULL. The error is returned by the input/output subsystem of the MQX Real-Time Operating System.
- *MFS_ERROR_UNKNOWN_FS_VERSION*
  - MFS was installed on a disk using the FAT32 file system, and the FAT32 version is incompatible with the MFS FAT32 version (version zero).
- *MFS_INSUFFICIENT_MEMORY*
  - MFS could not allocate memory for required structures.
- *MFS_NO_ERROR*
  - The function call was successful.
- *MFS_NOT_A_DOS_DISK*
  - The device, on which MFS is being installed is not a valid DOS device. The device must be formatted (by an input/output control command).
- *MFS_NOT_INITIALIZED*

— The MFS device name did not end with colon (*:*).

- *MFS_READ_FAULT*
  — The lower-level device driver could not read from the disk. The error is returned from the device, over which MFS is installed.
- *MFS_SECTOR_NOT_FOUND*
  — The error is returned from the device, over which, MFS is installed.
- *PGMR_INVALID_PARTITION*
  — The partition number specified was that of an invalid partition. The partition does not exist.

### Example

Install MFS on a RAM disk with no partitions.

```
/* Install the memory device: */
   error_code = _io_mem_install("mfsram:",
      NULL, MFS_format.BYTES_PER_SECTOR * RAMDISK_LENGTH1);
   if ( error_code != MQX_OK ) {
      printf("Error installing device.\nError: %d\n", error_code);
      _mqx_exit(1);
   }

   /* Open the device on which MFS will be installed: */
   dev_handle1 = fopen("mfsram:", 0);
   if ( dev_handle1 == NULL ) {
      printf("\nUnable to open RAM disk device");
      _task_block();
   }

   /* Install MFS:  */
   error_code = _io_mfs_install(dev_handle1, "MFS1:", 0);
   if ((error_code != MFS_NO_ERROR) &&
         (error_code  != MFS_NOT_A_DOS_DISK)) {
      printf("FATAL error while initializing: \n");
      _mqx_exit(1);
   } else {
      printf("Initialized MFS1%s\n");
   }
```

## 3.3 _io_mfs_uninstall

Uninstall MFS.

### Synopsis

```
uint32_t _io_mfs_uninstall(
       /*[IN] String that identifies the device driver   */
       /* to uninstall. Must be identical to the string  */
       /* that was used to install the MFS device driver */
       char *     identifier)
```

### Description

This function uninstalls the MFS device driver and frees the memory allocated for it. Before you call the function, you must close the MFS device driver by calling **fclose()**.

### Return Codes

Returns a **uint32_t** error code.

- *MFS_INVALID_PARAMETER*
  - The identifier passed to the function is invalid.
- *MFS_SHARING_VIOLATION*
  - There are files still open on the device, or the MFS device is still open.

### Example

```
error_code = _io_mfs_uninstall("MFS1:");
```

## 3.4 _io_part_mgr_install

Installs the partition manager device driver.

**Synopsis**

```
int32_t _io_part_mgr_install(
      /*[IN] Handle of the device on which to install */
      /* the partition manager */
      FILE_PTR    dev_fd,
      /*[IN] New name of the partition manager device */
      char *    identifier,
      /*[IN] Size of sectors in bytes on the lower level device */
      uint32_t     sector_size)
```

**Description**

This function initializes the partition manager device driver and allocates the memory for its internal structures.

The first parameter is the handle acquired by opening the lower-level device driver using **fopen()** (for example, `dev_fd = fopen("flashdisk",0))`.

The second parameter is the identifier, under which the partition manager is to be installed.

The third parameter is the sector size of the disk. If you specify zero, the partition manager queries the disk for the sector size. If the query fails, the partition manager uses a default sector size, as defined by *PMGR_DEFAULT_SECTOR_SIZE*. The default is 512 bytes.

**Errors**

- *PMGR_INSUF_MEMORY*
  - Partition manager could not allocate memory for its internal data.

**Example**

Install the partition manager as "PM:" and let it determine the sector size.

```
error_code = _io_part_mgr_install(dev_fd, "PM:", 0);
```

Obtain the handle to the partition manager without selecting a particular partition, i.e. with access to the whole underlying device.

```
pm_fd = fopen("PM1:",0);
```

Obtain the handle to the partition manager with the first partition selected, i.e. the read/write access is limited to the first partition.

```
part_fd = fopen("PM1:1",0);
```

## 3.5 _io_part_mgr_uninstall

Uninstalls the partition manager.

### Synopsis

```
int32_t _io_part_mgr_uninstall(
      /*[IN] Identifier string of the device */
      char *  identifier)
```

### Description

You must close the partition manager before you uninstall it. The first parameter is the same identifier that is used with **_io_part_mgr_install()**. All handles associated with a given partition manager have to be closed prior to calling the function. Otherwise, the function fails.

### Errors

- *IO_EOF*
  - Incorrect *identifier*.
- *IO_ERROR_DEVICE_BUSY*
  - There are still open handles associated with the partition manager instance.

### Example

```
error_code = _io_part_mgr_uninstall("PM:");
```

## 3.6 fclose

Closes the device or file.

**Synopsis**

```
int32_t _io_fclose(
      /* [IN] Stream to close (MFS) */
      /* or file pointer of the partition manager to close */
      FILE_PTR    file_ptr)
```

**Description**

This function frees the memory allocated for the given *FILE_PTR* (which was returned when the application called **fopen()** on a file). It also updates the date, time, and size of the file on the disk.

When the application no longer needs to use the device driver, it can close the device driver and uninstall it. The function **fclose()** is used to close the device driver if the device driver *FILE_PTR* is passed as a parameter (as opposed to a regular file *FILE_PTR*). The function fails if any files are still open on the device.

**Return Codes for MFS**

- *IO_EOF*
  - *file_ptr* was invalid.
- *MFS_SHARING_VIOLATION*
  - Files are open on the device.

**Example: MFS**

See **fopen()**.

**Example: Partition Manager Device Driver**

```
pmgr_fd_ptr = fopen("PM:", NULL);
...
...
/* End of application. */
fclose(pmgr_fd_ptr);
_io_part_mgr_uninstall("PM:");
```

## 3.7 fopen

Opens the device or file.

### Synopsis

```
FILE_PTR _io_fopen(
      /*[IN] Name of the device or file to open      */
      /* Must be identical to the name that was used */
      /* to install the device driver */
      char * open_type_ptr,
      /*[IN] I/O parameter to pass to device initialization  */
      /* This parameter is for extra parameters. It is only  */
      /* used when opening files                             */
      /* Must be NULL for the partition manager               */
      char * open_mode_ptr)
```

### Description

This function opens the specified device driver for MFS or the partition manager. You must install the device driver before you call the function. Opening the device returns a *FILE_PTR* for the device that can be used in input/output control commands (see Section 3.8, “ioctl”).

The first time **fopen()** is called on the device driver, it opens the device driver. Each subsequent call is used to open a file. This means that you must first call **fopen()** with the device name (just once to open the device) with NULL as the *open_mode_ptr*, and then every other call will be to open a file. Each of these other calls should include the device name, along with a specific flag for the *open_mode_ptr*.

The function is also used to open files on the device. Opening a file returns a *FILE_PTR* for that file. This is used to read and write to the file. All the standard read and write functions work on files such as **write()**, **read()**, **fscanf()**, **fputc()**, and so on. All the other standard input/output functions also work on devices, for example, **fseek()** changes the position within a file.

Here is a list of the standard MQX functions that can be used:

**_io_clearerr()**, **_io_fclose()**, **_io_feof()**, **_io_ferror()**, **_io_fflush()**, **_io_fgetc()**, **_io_fgetline()**, **_io_fgets()**, **_io_fopen()**, **_io_fprintf()**, **_io_fputc()**, **_io_fputs()**, **_io_scanf()**, **_io_fseek()**, **_io_fstatus()**, **_io_ftell()**, **_io_fungetc()**, **_io_ioctl()**, **_io_printf()**, **_io_putc()**, **_io_read()**, **_io_scanf()**, **_io_sprintf()**, **_io_sscanf()**, **_io_vprintf()**, **_io_vfprintf()**, **_io_vsprintf()**, **_io_write()**.

To open a file, you must pass the name of the device followed by the name of the file. To open the file *data.txt* in the current directory:

```
fd_ptr = fopen(“MFS1:data.txt”, “w”);
To open the file March2000results.data in the MFS1:\data\march directory:
fd_ptr = fopen(“MFS1:\data\march\March2000results.data”);
```

Here is a list of different options for the second parameter:

| Option | Description |
|---|---|
| NULL | Open the device (either MFS or the partition manager). |
| "w" | Open a new file in “write-only” mode; overwrite an existing file. |
| "w+" | Open a new file in “read-write” mode; overwrite an existing file. |
| "r" | Open an existing file in “read-only” mode. |
| "r+" | Open an existing file in “read-write” mode. |
| "a" | Open a file at EOF in “write” mode; create the file if it does not exist. |
| "a+" | Open a file at EOF in “read-write” mode; create the file if it does not exist. |
| "n" | Open a new file in “write-only” mode; do nothing if the file already exists. |
| "n+" | Open a new file in “read-write” mode; do nothing if the file already exists. |
| "x" | Create a temporary file. MFS or the partition manager assigns the filename and appends it to the end of the *char ** passed into the function. This means that you must have at least 15 bytes of unused space at the end of the *open_type_ptr* string and its last used byte cannot be a directory separator. |

### Returns

Returns a *FILE_PTR* to the new file or to the device on success.

Returns NULL on failure and calls **_task_set_error()** to set the task error code.

### Example: MFS

Open the MFS device driver and open a file on the device.

```
char buffer[100] = "This a test file";
char buffer2[100];
/* Open the MFS device driver: */
mfs_fd_ptr = fopen("MFS1:", NULL);
if (mfs_fd_ptr == NULL)  {
    printf("Error opening the MFS device driver!");
    _mqx_exit(1);
}
/* Open file on disk in the current directory and write to it: */
fd_ptr = fopen("MFS1:myfile.txt", "w+");
write(fd_ptr, buffer, strlen(buffer));
read(fd_ptr, buffer2, strlen(buffer));

/* Close the file: */
error_code = fclose(fd_ptr);

/* Open other files, create directories, and so on. */

/* The application has done all it needs. */
/* Close the MFS device driver and uninstall it: */
error_code = fclose(mfs_fd_ptr);
if (!error_code) {
```

```
        error_code = _io_mfs_uninstall("MFS1:");
    } else if (error_code == MFS_SHARING VIOLATION) {
    printf("There are open files on the device. Call fclose on their
           handles before attempting to fclose the device driver");
     }
```

### Example: Partition Manager Device Driver

The example assumes that the partition manager is already installed.

```
pmgr_fd_ptr = fopen("PM:",NULL);
```

## 3.8 ioctl

Issues a control command.

**Synopsis**

```
int32_t _io_ioctl(
      /*[IN] Stream to perform the operation on */
      FILE_PTR    file_ptr,
      /*[IN] I/O control command */
      uint32_t     cmd,
      /*[IN] I/O control-command parameters */
      uint32_t * param_ptr)
```

**Description**

The first parameter is a *FILE_PTR*, returned by calling **fopen()** for the device driver, which can either be the handle of a specific file, or the handle of the device driver itself. It varies depending on which command is used. The third parameter is a *uint32_t **. Depending upon the input/output control command, it is usually a different kind of pointer cast to a *uint32_t **. For example, it might be a *char **, a pointer to a structure, or even a NULL pointer.

| | |
|---|---|
| **CAUTION** | *MFS _io_ioctl* calls do not always follow the standard of returning data as the third parameter and the result as the function returns. In many cases, data is returned as the function returns, and an error code is not available. You must ensure that the *_io_ioctl* call is used correctly for the specified control command. |

### 3.8.1 Input/Output Control Commands for MFS

Together with the MQX input/output control commands, MFS also includes the following input/output control commands.

#### 3.8.1.1 IO_IOCTL_BAD_CLUSTERS

This command gets the number of bad clusters on the drive.

```
num_of_bad_clusters = ioctl(mfs_fd_ptr,
                            IO_IOCTL_BAD_CLUSTERS,
                            NULL);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. The *mfs_fd_ptr* must correspond to the disk, on which the bad clusters are to be counted. The third parameter is a NULL pointer.

#### 3.8.1.2 IO_IOCTL_CHANGE_CURRENT_DIR

This command changes the current directory.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_CHANGE_CURRENT_DIR,
                  (uint32_t *) pathname);
```

If *pathname* begins with a directory separator, it is assumed that *pathname* represents the complete directory name. If *pathname* does not begin with a directory separator, *pathname* is assumed to be relative to the current directory. The third parameter is a *char* * (to a directory name) cast to a *uint32_t* *.

The directory path must exist for the change to succeed.

**Errors**

- *MFS_INVALID_LENGTH_IN_DISK_OPERATION*
  - — Path name is too long. The full path name, including the filename, cannot be any longer than 260 characters.

**Example**

```
char pathname = "\\docs";
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_CHANGE_CURRENT_DIR,
                     (uint32_t *) pathname);
```

### 3.8.1.3 IO_IOCTL_CREATE_SUBDIR

This command creates a subdirectory in the current directory.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_CREATE_SUBDIR,
                   (uint32_t *) “\temp\newdir”);
```

A path name can be specified to create the subdirectory in a different directory. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver corresponding to the disk on which to operate. The third parameter is a *char* * (to a directory name) cast to a *uint32_t* *

All directories in the path, except the last one, must exist. The last directory in the path must not exist as either a directory or a file.

**Errors**

- *MFS_CANNOT_CREATE_DIRECTORY*
  - — There was an error creating the subdirectory.

### 3.8.1.4 IO_IOCTL_DEFAULT_FORMAT

This command formats the drive by using default parameters.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_DEFAULT_FORMAT,
                  NULL);
```

The command deletes all files and subdirectories on the drive. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver, which corresponds to the disk on which to operate. The default parameters are:

- *PHYSICAL_DRIVE* = 0x80
- *MEDIA_DESCRIPTOR* = 0xf8
- *BYTES_PER_SECTOR* = device sector size
- *SECTORS_PER_TRACK* = 0x00

- *NUMBER_OF_HEADS* = 0x00
- *NUMBER_OF_SECTORS* = number of device sectors - RESERVED_SECTORS
- *HIDDEN_SECTORS* = 0
- *RESERVED_SECTORS* = 1 if *NUMBER_OF_SECTORS* < 2097152, 32 otherwise

**Errors**

- *MFS_SHARING_VIOLATION*
  - Some files are open on the drive.

**Example**

```
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_FORMAT,NULL);
```

### 3.8.1.5 IO_IOCTL_DELETE_FILE

This command deletes a file on the disk. Wildcard characters are not valid in the filename.

```
error_code = _io_ioctl(mfs_fd_ptr,
                       IO_IOCTL_DELETE_FILE,
                       (uint32_t *) “filename”);
```

The *mfs_fd_ptr* is the *FILE_PTR* returned from **fopen()** that opened the MFS device. The third parameter points to a filename which can include a path (for example *\backup\oldfiles\myfile.txt*). Long filenames and long path names are supported. The file must reside on the drive that corresponds to *mfd_fd_ptr*.

Any currently open handles to this file become invalid, that is, subsequent file operations using a file handle of a deleted file result in an error.

**Errors**

- *MFS_OPERATION_NOT_ALLOWED*

### 3.8.1.6 IO_IOCTL_FAT_CACHE_OFF

### 3.8.1.7 IO_IOCTL_FAT_CACHE_ON

Depricated: use *IO_IOCTL_SET_FAT_CACHE_MODE*.

Set the FAT cache mode to write through (OFF) or write back (ON).

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_FAT_CACHE_OFF,
                   NULL);
```

The first parameter is the *FILE_PTR* of the MFS device driver that corresponds to the disk on which the operation is to take place. The third parameter is a NULL pointer.

### 3.8.1.8 IO_IOCTL_FIND_FIRST_FILE

### 3.8.1.9 IO_IOCTL_FIND_NEXT_FILE

This command searches for a file on the disk.

If a file is found, use the input/output control command, *IO_IOCTL_FIND_NEXT_FILE*, to keep searching for files with the same criteria as the first. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. The third parameter is a pointer to the *MFS_SEARCH_PARAM* structure cast to a *uint32_t **. The three fields of the structure must be initialized. See structure definitions for details.

The third parameter for the *IO_IOCTL_FIND_NEXT_FILE* is a pointer to the *MFS_SEARCH_DATA* structure used in the *IO_IOCTL_FIND_FIRST_FILE* command. It must be cast to a **uint32_t ***.

The filename can include wildcard search characters.

When searching for long filenames, only one asterisk (*) is allowed. Everything after the one asterisk (*) is assumed to be wildcard characters.

When searching for files, the file path search string that is passed in the *MFS_SEARCH_PARAM* structure is used. Therefore, it must not be freed or changed if you plan to subsequently use IO_IOCTL_FIND_NEXT_FILE.

The search criteria for the attribute field of the *MFS_SEARCH_PARAM* structure is defined in the following table:

| Attribute: | Value: | Return these types of entries: |
|---|---|---|
| MFS_SEARCH_NORMAL | 0x00 | Non-hidden non-system files and directories |
| MFS_SEARCH_READ_ONLY | 0x01 | Read only files and directories |
| MFS_SEARCH_HIDDEN | 0x02 | Hidden files and directories |
| MFS_SEARCH_SYSTEM | 0x04 | System files and directories |
| MFS_SEARCH_VOLUME | 0x08 | Volume label only |
| MFS_SEARCH_SUBDIR | 0x10 | Non-hidden non-system directories |
| MFS_SEARCH_ARCHIVE | 0x20 | Archive files and directories |
| MFS_SEARCH_EXCLUSIVE | 0x40 | Match exactly all remaining attributes |
| MFS_SEARCH_ANY | 0x80 | All files and directories |

The search bit mask can be a combination of all search attributes. The evaluation of the bit mask is done in the following order:

1. If mask includes MFS_SEARCH_ANY, then all disk entries match.
2. If mask includes MFS_SEARCH_VOLUME, then only the volume label entry matches.
3. If mask includes MFS_SEARCH_EXCLUSIVE, then there must be an exact match of the remaining attributes.

4. If mask is MFS_SEARCH_NORMAL, then all non-system, non-hidden files and directories, match.
5. If mask is MFS_SEARCH_SUBDIR, then all non-system, non-hidden directories, match.
6. Otherwise mask must be subset of disk entry attributes to produce a match.

The search results are written into the *MFS_SEARCH_DATA* structure addressed by *search_data*. If the *IO_IOCTL_FIND_NEXT* command is used, its results are written over the previous results.

The results of file searches are written into this data structure.

```
typedef struct MFS_search_data
{
   unsigned char RESERVED[25];
   unsigned char ATTRIBUTE;
   uint16_t   TIME;
   uint16_t  DATE;
   uint32_t  FILE_SIZE;
   char     NAME[13];
} MFS_SEARCH_DATA,
  * MFS_SEARCH_DATA_PTR;
```

See also *IO_IOCTL_FIND_FIRST_FILE*, *IO_IOCTL_FIND_NEXT_FILE* and *IO_IOCTL_GET_DATE_TIME*.

The results of *IO_IOCTL_FIND_FIRST_FILE* and *IO_IOCTL_FIND_NEXT_FILE* are written into a data structure of type *MFS_SEARCH_DATA*.

***MFS_SEARCH_DATA Fields***

**ATTRIBUTE**

File entry attribute byte.

**TIME**

File entry time, as described in *IO_IOCTL_GET_DATE_TIME*.

**DATE**

File entry date, as described in *IO_IOCTL_GET_DATE_TIME*.

**FILE_SIZE**

Size of the file in bytes.

**NAME[13]**

ASCII name of the file in the format *filename.filetype*.

**Errors**

- *MFS_INVALID_MEMORY_BLOCK_ADDRESS*
  - The *MFS_SEARCH_DATA_PTR* in the *MFS_SEARCH_PARAM* is invalid.

### Example

```
List all files and subdirectories in a directory.
MFS_SEARCH_DATA      search_data;
MFS_SEARCH_PARAM     search;
char                 filepath = “*.*”;

search.ATTRIBUTE = MFS_SEARCH_ANY;
search.WILDCARD = filepath;
search.SEARCH_DATA_PTR = &search_data;

error_code = ioctl(mfs_fd_ptr, IO_IOCTL_FIND_FIRST_FILE,
                     (uint32_t *) &search);
while (error_code == MFS_NO_ERROR) {
      printf ("%-12.12s  %6lu %02lu-%02lu-%04lu  %02lu:%02lu:%02lu
         \n", search_data.NAME, search_data.FILE_SIZE,
         (uint32_t)(search_data.DATE & MFS_MASK_MONTH) >>
         MFS_SHIFT_MONTH,
         (uint32_t)(search_data.DATE & MFS_MASK_DAY)   >>
         MFS_SHIFT_DAY,
         (uint32_t)((search_data.DATE & MFS_MASK_YEAR) >>
         MFS_SHIFT_YEAR) + 1980,
         (uint32_t)(search_data.TIME & MFS_MASK_HOURS)   >>
         MFS_SHIFT_HOURS,
         (uint32_t)(search_data.TIME & MFS_MASK_MINUTES) >>
         MFS_SHIFT_MINUTES,
         (uint32_t)(search_data.TIME & MFS_MASK_SECONDS) << 1);

       error_code = ioctl(mfs_fd_ptr, IO_IOCTL_FIND_NEXT_FILE,
                           (uint32_t *) &search_data);
   }
```

### 3.8.1.10 IO_IOCTL_FIND_NEXT_FILE

See *IO_IOCTL_FIND_FIRST_FILE*.

### 3.8.1.11 IO_IOCTL_FLUSH_FAT

If the file allocation table has been modified and not yet written to a disk, it will be written to a disk.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_FLUSH_FAT,
                   NULL);
```

The first parameter is the *FILE_PTR* of the MFS device driver that corresponds to the disk on which the operation is to take place. The third parameter is a NULL pointer.

### 3.8.1.12 IO_IOCTL_FORMAT

This command formats the drive according to the given specifications.

```
error_code = ioctl(mfs_fd_ptr,
                    IO_IOCTL_FORMAT,
                   (uint32_t *) &format_struct);
```

The command deletes all files and subdirectories on the drive. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver, which corresponds to the disk on which to operate. The third parameter is a pointer to the *MFS_IOCTL_FORMAT_PARAM* structure cast to the *uint32_t **. The only field in the *MFS_IOCTL_FORMAT_PARAM* structure that must be initialized is the *FORMAT_PTR* field. See the structure descriptions for details.

**Errors**

- *MFS_SHARING_VIOLATION*
  - — Some files are open on the drive.

**Example**

```
MFS_IOCTL_FORMAT_PARAM     format_struct;
MFS_FORMAT_DATA MFS_format =
{
   /* PHYSICAL_DRIVE;    */   PHYSICAL_DRI,
   /* MEDIA_DESCRIPTOR;  */   MEDIA_DESC,
   /* BYTES_PER_SECTOR;  */   BYTES_PER_SECT,
   /* SECTORS_PER_TRACK; */   SECTS_PER_TRACK,
   /* NUMBER_OF_HEADS;   */   NUM_OF_HEADS,
   /* NUMBER_OF_SECTORS; */   1000,   /* depends on drive */
   /* HIDDEN_SECTORS;    */   HIDDEN_SECTS,
   /* RESERVED_SECTORS;  */   RESERVED_SECTS
};
format_struct.FORMAT_PTR = &MFS_format;
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_FORMAT,
                     (uint32_t *) &format_struct);
```

### 3.8.1.13 IO_IOCTL_FORMAT_TEST

This command formats the drive and counts the bad clusters on a disk.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_FORMAT_TEST,
                   (uint32_t *) &format_struct);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver, which corresponds to the device on which to operate. The third parameter is a pointer to the *MFS_IOCTL_FORMAT_PARAM* structure cast to the *uint32_t **. Both fields of the *MFS_IOCTL_FORMAT_PARAM* structure must be initialized (*FORMAT_PTR* and *COUNT_PTR*). See structure descriptions for details.

**Errors**

- MFS_SHARING_VIOLATION
  - — Files are open on the drive.

**Example**

```
uint32_t                  bad_cluster_count;
MFS_IOCTL_FORMAT_PARAM   format_struct;
MFS_FORMAT_DATA          MFS_format =
{
```

```
   /* PHYSICAL_DRIVE;    */   PHYSICAL_DRI,
   /* MEDIA_DESCRIPTOR;  */   MEDIA_DESC,
   /* BYTES_PER_SECTOR;  */   BYTES_PER_SECT,
   /* SECTORS_PER_TRACK; */   SECTS_PER_TRACK,
   /* NUMBER_OF_HEADS;   */   NUM_OF_HEADS,
   /* NUMBER_OF_SECTORS; */   1000,    /* depends on disk */
   /* HIDDEN_SECTORS;    */   HIDDEN_SECTS,
   /* RESERVED_SECTORS;  */   RESERVED_SECTS
};
format_struct.FORMAT_PTR = &MFS_format;
format_struct.COUNT_PTR = &bad_cluster_count;
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_FORMAT,
                     (uint32_t *) &format_struct);
if (!error_code)
printf("The count of bad clusters is: %d\n", bad_cluster_count);
```

### 3.8.1.14 IO_IOCTL_FREE_CLUSTERS

### 3.8.1.15 IO_IOCTL_FREE_SPACE

This command gets the count of free space in clusters or in bytes.

```
space = ioctl(mfs_fd_ptr,
              IO_IOCTL_FREE_SPACE,
              NULL);

or

result = ioctl(mfs_fd_ptr,
              IO_IOCTL_FREE_SPACE,
              &space_64);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. It should correspond to the disk on which the free space is to be calculated. The third parameter is an optional pointer to uint64_t which is filled with 64-bit value representing free space in bytes.

Please note that the second form of usage is required to obtain the correct value for large drives with capacity of 4GB or more. Alternatively, a combination of *IO_IOCTL_FREE_CLUSTERS* and *IO_IOCTL_GET_CLUSTER_SIZE* may be used to determine the free space size of the drive in bytes by using long (64-bit) arithmetic.

### 3.8.1.16 IO_IOCTL_GET_CLUSTER_SIZE

This command gets the size of clusters in bytes.

```
result = ioctl(mfs_fd_ptr,
              IO_IOCTL_GET_CLUSTER_SIZE,
              &cluster_size);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. It should correspond to the disk for which the cluster size should be returned. The third parameter is a pointer to a *uint32_t* * to pre-allocated space in which to store the cluster size.

### 3.8.1.17 IO_IOCTL_GET_CURRENT_DIR

This command gets the path name of the current directory on the MFS device.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_GET_CURRENT_DIR,
                   (uint32_t *) pathname);
```

The drive and drive separator are not included in the filename (for example, "d:" is not returned). The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver corresponding to the disk on which to operate. The third parameter is a *char **, to the -allocated space in which to store the current directory, cast to a *uint32_t **.

**Example**

```
char pathname[261];
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_GET_CURRENT_DIR,
                     (uint32_t *) pathname);
printf("The current directory is: %s\n", pathname);
```

### 3.8.1.18 IO_IOCTL_GET_DATE_TIME

This command gets the current date and time associated with the file.

```
error_code = ioctl(fd_ptr,
                   IO_IOCTL_GET_DATE_TIME,
                   (uint32_t *) &date);
```

The first parameter is the **FILE_PTR** of the file for which the date or time is to be retrieved. The third parameter is a pointer to a **MFS_DATE_TIME_PARAM** structure that is cast to a **uint32_t ***. Both fields of the structure must be filled in. See structure definitions for details.

The bits of the date and time words are defined as follows:

| Time word | | | Date word | |
|---|---|---|---|---|
| **Bits** | **Meaning** | | **Bits** | **Meaning** |
| 4 – 0 | 0 – 29, 2 second increments | | 4 – 0 | 1 – 31 days |
| 10 – 5 | 0 – 59 minutes | | 8 – 5 | 1 – 12 month |
| 15 – 11 | 0 – 23 hours | | 15 – 9 | 0 – 119 year (1980 – 2099) |

**Example**

```
uint32_t              error_code;
uint16_t              date_word, time_word;
MFS_DATE_TIME_PARAM  date;

date.DATE_PTR = &date_word;
date.TIME_PTR = &time_word;
```

```
error_code = ioctl(fd_ptr, IO_IOCTL_GET_DATE_TIME,
                    (uint32_t *) &date);
if (!error_code )
printf ("%02lu-%02lu-%04lu  %02lu:%02lu:%02lu \n",
         (uint32_t)(date_word & MFS_MASK_MONTH) >> MFS_SHIFT_MONTH,
         (uint32_t)(date_word & MFS_MASK_DAY)   >> MFS_SHIFT_DAY,
         (uint32_t)((date_word & MFS_MASK_YEAR) >> MFS_SHIFT_YEAR)
         + 1980,
         (uint32_t)(time_word.TIME & MFS_MASK_HOURS)   >>
         MFS_SHIFT_HOURS,
         (uint32_t)(time_word.TIME & MFS_MASK_MINUTES) >>
         MFS_SHIFT_MINUTES,
         (uint32_t)(time_word.TIME & MFS_MASK_SECONDS) << 1);
```

### 3.8.1.19 IO_IOCTL_GET_DEVICE_HANDLE

This command gets the handle of the low-level device which this instance of the file system is operating on.

```
result = ioctl(mfs_fd_ptr,
          IO_IOCTL_GET_DEVICE_HANDLE,
          &handle);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. The third parameter is a pointer to a *FILE_PTR* (cast to a *uint32_t* *) which points to pre-allocated space in which to store the device handle.

### 3.8.1.20 IO_IOCTL_GET_FAT_CACHE_MODE

### 3.8.1.21 IO_IOCTL_SET_FAT_CACHE_MODE

This command gets or sets the current mode of the FAT cache.

```
result = ioctl(mfs_fd_ptr,
              IO_IOCTL_GET_FAT_CACHE_MODE,
              &mode);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. The third parameter is a *_mfs_cache_policy* pointer (cast to a *uint32_t* *) which points to pre-allocated space in which to store (when using get) or obtain (when using set) the FAT cache mode.

### 3.8.1.22 IO_IOCTL_GET_FILE_ATTR

### 3.8.1.23 IO_IOCTL_SET_FILE_ATTR

This command gets or sets the attribute byte from a file on disk.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_GET_FILE_ATTR,
                   (uint32_t *) &attr);
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_SET_FILE_ATTR,
```

```
                    (uint32_t *) & attr);
```

An application cannot set the volume or directory bits of the attribute **char**. The first parameter is the **FILE_PTR** of the MFS device driver that corresponds to the disk on which the file whose attributes are to be read or written is located. The third parameter is a pointer to a **MFS_FILE_ATTR_PARAM** structure that is cast to a **uint32_t ***. Both fields of the structure must be filled in. See the structure definitions for details.

**Example**

```
MFS_FILE_ATTR_PARAM  attr;
uint32_t  error_code;
char     filepath = “\temp\myfile.txt”;
unsigned char attribute;

attr.ATTRIBUTE_PTR = &attribute;
attr.PATHNAME = filepath;

/* Get the attribute: */
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_GET_FILE_ATTR,
                    (uint32_t *) &attr);

if (error_code == MFS_NO_ERROR) {
    printf ("Attributes of %s: %s%s%s%s%s%s\n",
       filepath,
       (attribute & MFS_ATTR_READ_ONLY) ? "R/O ":"",
       (attribute & MFS_ATTR_HIDDEN_FILE) ? "HID ":"",
       (attribute & MFS_ATTR_SYSTEM_FILE) ? "SYS ":"",
       (attribute & MFS_ATTR_VOLUME_NAME) ? "VOL ":"",
       (attribute & MFS_ATTR_DIR_NAME) ? "DIR ":"",
       (attribute & MFS_ATTR_ARCHIVE) ? "ARC ":"");
}
/* Set file’s attributes: */
if (!error_code) {
attribute = MFS_ATTR_READ_ONLY | MFS_ATTR_HIDDEN_FILE;
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_SET_FILE_ATTR,
                    (uint32_t *) & attr);
}
```

### 3.8.1.24 IO_IOCTL_GET_LFN

This command gets the long filename where the path name is in 8.3 representation.

*error_code* = ioctl(*mfs_fd_ptr*,
IO_IOCTL_GET_LFN,
(uint32_t *) &*lfn_struct*);

The first parameter is the *FILE_PTR* of the MFS device driver that corresponds to the disk on which the operation is to take place. The third parameter is the *char** to the path name of the of file which we want the long filename of. It is cast to the *uint32_t **.

**Example**

```
MFS_GET_LFN_STRUCT   lfn_struct;
char                 lfn[FILENAME_SIZE + 1];
```

```
char                filepath = "\\temp\longfi~1.txt";
uint32_t             error_code;

lfn_struct.PATHNAME = filepath;
lfn_struct.LONG_FILENAME = lfn;

error_code = ioctl(mfs_fd_ptr, IO_IOCTL_GET_LFN,
                    (uint32_t *) &lfn_struct);

if (!error_code)  {
    printf("%s\n", lfn);
}
```

### 3.8.1.25 IO_IOCTL_GET_VOLUME

This command gets the volume label.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_GET_VOLUME,
                   (uint32_t *) label);
```

The first parameter is the **FILE_PTR** of the MFS device driver that corresponds to the disk on which the operation is to take place. The third parameter is a **char *** to an allocated space with 12 free bytes in which the volume label will be written. It is cast into a **uint32_t ***.

**Example**

```
charlabel[12];
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_GET_VOLUME,
                    (uint32_t *) label);
if (!error_code) {
printf("The volume label is: %d\n", label);

/* Now set the volume label */
strcpy(label, "newlabel");
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_SET_VOLUME,
                           (uint32_t *) label);
}
```

### 3.8.1.26 IO_IOCTL_GET_WRITE_CACHE_MODE

### 3.8.1.27 IO_IOCTL_SET_WRITE_CACHE_MODE

This command gets or sets the current mode of the data and directory caches.

```
result = ioctl(mfs_fd_ptr,
               IO_IOCTL_GET_WRITE_CACHE_MODE,
               &mode);
```

The parameter *mfs_fd_ptr* is the **FILE_PTR** returned when **fopen()** was called on the MFS device driver. The third parameter is a _mfs_cache_policy pointer (cast to a **uint32_t ***) which points to a pre-allocated space in which to store (when using get) or obtain (when using set) the mode of the write caches.

### 3.8.1.28 IO_IOCTL_LAST_CLUSTER

This command gets the number of clusters on a drive.

```
last_cluster = ioctl(mfs_fd_ptr,
                     IO_IOCTL_LAST_CLUSTER,
                     NULL);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. It should correspond to the disk, on which the free space is to be calculated. The third parameter is a NULL pointer.

### 3.8.1.29 IO_IOCTL_REMOVE_SUBDIR

This command removes a the subdirectory in the current directory.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_REMOVE_SUBDIR,
                   (uint32_t *) “\temp\deldir”);
```

A path name can be specified to remove the subdirectory in a different directory. The subdirectory must be empty and cannot be the current directory or the root directory. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver corresponding to the disk on which to operate. The third parameter is the *char* * (to a directory name) cast into the *uint32_t* *.

**Errors**

- *MFS_ATTEMPT_TO_REMOVE_CURRENT_DIR*
  - The directory specified is the current directory. No changes took place.

### 3.8.1.30 IO_IOCTL_RENAME_FILE

This command renames a file or moves a file if path names are specified.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_RENAME_FILE,
                   (uint32_t *) &rename_struct);
```

No wildcard characters are allowed in the path names. The parameter *mfs_fd_ptr* is the *FILE_PTR* returned, when **fopen()** was called on the MFS device driver corresponding to the drive on which to operate. The third parameter is a pointer to the *MFS_RENAME_PARAM* structure cast to the *uint32_t* *. Both fields in this structure must be filled out. See structure definitions for details.

A file is moved if the directory paths are different and the file names are the same. A file is renamed if the directory paths are the same and the file names are different.

A directory can be renamed, but cannot be moved.

**Example**

```
MFS_RENAME_PARAM  rename_struct;
char              oldpath[PATHNAME_SIZE + 1],
                  newpath[PATHNAME_SIZE + 1];
uint32_t           error_code;
```

```
rename_struct.OLD_PATHNAME = oldpath;
rename_struct.NEW_PATHNAME = newpath;

/* Rename a file: */
strcpy(oldpath, “myfile.txt”);
strcpy(newpath, “myfile.bak”);
error_code = ioctl(mfs_fd_ptr, IO_IOCTL_RENAME_FILE,
                    (uint32_t *) &rename_struct);

/* Move the file: */
if (!error_code) {
strcpy(oldpath, “myfile.bak”);
       strcpy(newpath, “\temp\temp.tmp”);
       error_code = ioctl(mfs_fd_ptr, IO_IOCTL_RENAME_FILE,
                           (uint32_t *) &rename_struct);
}
```

### 3.8.1.31 IO_IOCTL_SET_DATE_TIME

This command sets the time and date of an open file.

```
error_code = ioctl(fd_ptr,
                   IO_IOCTL_SET_DATE_TIME,
                   (uint32_t *) &date);
```

The first parameter is the *FILE_PTR* of the file for which to set the date. The third parameter is a pointer to the *MFS_DATE_TIME_PARAM* structure that is cast to the *uint32_t **. Both fields of the structure must be filled in. See the structure definitions for more information.

**Example**

See *IO_IOCTL_GET_DATE_TIME* for details.

```
MFS_DATE_TIME_PARAM  date_time;
   uint32_t             error_code;
   uint16_t             date_word, time_word;

date.DATE_PTR = &date_word;
date.TIME_PTR = &time_word;

error_code = ioctl(fd_ptr, IO_IOCTL_GET_DATE_TIME,
                    (uint32_t *) &date);
```

### 3.8.1.32 IO_IOCTL_SET_FAT_CACHE_MODE

See *IO_IOCTL_GET_FAT_CACHE_MODE*.

### 3.8.1.33 IO_IOCTL_SET_FILE_ATTR

See *IO_IOCTL_GET_FILE_ATTR*.

#### 3.8.1.34 IO_IOCTL_SET_VOLUME

This command sets the volume label.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_SET_VOLUME,
                   (uint32_t *) label);
```

The first parameter is the *FILE_PTR* of the MFS device driver that corresponds to the disk on which the operation is to take place. The third parameter is the *char* * to the new volume name to be set with a maximum of 11 characters. It is cast to the *uint32_t *.*

#### 3.8.1.35 IO_IOCTL_SET_WRITE_CACHE_MODE

See *IO_IOCTL_GET_WRITE_CACHE_MODE*.

#### 3.8.1.36 IO_IOCTL_TEST_UNUSED_CLUSTERS

This command tests the unused clusters on the drive for bad clusters.

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_TEST_UNUSED_CLUSTERS,
                   &count_of_unused_clusters);
```

The parameter *mfs_fd_pt*r is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver corresponding to the drive, on which to test the unused clusters. The third parameter is the *uint32_t* * to a variable, in which the count of bad clusters is stored. The bad clusters are marked in the file allocation table so that they will not be used to store data.

#### 3.8.1.37 IO_IOCTL_WRITE_CACHE_OFF

#### 3.8.1.38 IO_IOCTL_WRITE_CACHE_ON

Depricated: use *IO_IOCTL_SET_WRITE_CACHE_MODE*.

Set the data and directory cache modes to write through (OFF) or write back (ON).

```
error_code = ioctl(mfs_fd_ptr,
                   IO_IOCTL_WRITE_CACHE_OFF,
                   NULL);
```

The parameter *mfs_fd_ptr* is the *FILE_PTR* returned when **fopen()** was called on the MFS device driver. The third parameter is a NULL pointer.

### 3.8.2 Input/Output Control Commands for the Partition Manager Device Driver

In addition to the MQX input/output control commands, the partition manager device driver includes the following.

### 3.8.2.1 IO_IOCTL_CLEAR_PARTITION

This command removes a partition from the disk.

The third **ioctl()** parameter is a pointer to the *uint32_t* variable and contains the number of the partition to remove. This IOCTL call is valid only if no partition is currently selected, i.e. the handle allows for access to the whole underlying device.

**Example**

Remove the third partition from the disk.

```
uint32_t   part_num;
part_num = 3;
error_code = ioctl(pmgr_fd_ptr, IO_IOCTL_CLEAR_PARTITION,
                    &part_num);
```

### 3.8.2.2 IO_IOCTL_GET_PARTITION

This command gets partition information to the disk.

The third **ioctl()** parameter is the *PMGR_PART_INFO_STRUCT* pointer that is cast to *uint32_t **. The only field in the structure that must be filled in is the *SLOT* field. It must contain a value between zero and four and represents the partition number for which information is requested. If the SLOT field is zero then information about currently selected partition is retrieved. The other fields are overwritten with the retrieved data. *HEADS*, *CYLINDERS*, and *SECTORS* are set to zero, because such information cannot be retrieved from the disk.

### 3.8.2.3 IO_IOCTL_SET_PARTITION

This command sets partition information to the disk.

The third **ioctl()** parameter is the *PMGR_PART_INFO_STRUCT* pointer that is cast to *uint32_t **.

```
typedef struct pmgr_part_info_struct
      {
         /* Partition slot (1 to 4) */
         unsigned char SLOT;
         /* Heads per Cylinder */
         unsigned char HEADS;
         /* Sectors per head */
         unsigned char SECTORS;
         /* Cylinders on the device */
         uint16_t   CYLINDERS;
         /* Partition type (0 not used, 1 FAT 12 bit, 4 FAT 16 bit, */
         /* 5 extended, 6 huge - DOS 4.0+, other = unknown OS) */
          unsigned char TYPE;
         /* Start sector for partition, relative to beginning of disk */
         uint32_t   START_SECTOR;
         /* Partition length in sectors */
         uint32_t   LENGTH;
      } PMGR_PART_INFO_STRUCT, * PMGR_PART_INFO_STRUCT_PTR;
```

The SLOT field must be filled in with the partition number to set.

The *HEADS*, *SECTORS*, and *CYLINDERS* fields are optional. They represent data that the partition manager uses to write the partition, but the data is used only by MS-DOS operating systems. Because Microsoft Windows does not use the fields on the disk, fill in the fields only if the disk is to be used with the MS-DOS operating system.

The *TYPE* field must be set to one of the following. Types that are marked with + are recommended when you create a partition.

| | | |
|---|---|---|
| + | PMGR_PARTITION_FAT_12_BIT | |
| | PMGR_PARTITION_FAT_16_BIT | Old FAT16 (MS-DOS 3.3 and previous ) |
| | PMGR_PARTITION_HUGE | Modern FAT16 (MS-DOS 3.3 and later) |
| | PMGR_PARTITION_FAT32 | Normal FAT32 |
| + | PMGR_PARTITION_FAT32_LBA | FAT32 with LBA |
| + | PMGR_PARTITION_HUGE_LBA | FAT16 with LBA |

The *START_SECTOR* field must be filled in. It is the physical sector on the device where the partition should start. For the first partition, is it generally sector 32 (for FAT32) or sector one (for FAT16 and FAT12). For partitions other than the first, it is the next sector after the end of the previous partition. You can leave unused sectors between partition, but they amount to wasted space.

The *LENGTH* field must be filled in. It contains the length in sectors of the new partition that is to be created.

This IOCTL call is valid only if no partition is currently selected, i.e. the handle allows for access to the whole underlying device and there is only a single open handle to the partition manager instance. This is to prevent possible inconsistence of data if more than one handle to the partition manager exists.

The partition manager checks validity of the partition table before writing it to the device. It is thus impossible to create a partition which overlaps another partition. Partitions which would collide with the new one have to be removed first.

**Example**

Create two partitions on a disk. The example assumes that the partition manager is installed and open.

```
PMGR_PART_INFO_STRUCT   part_info;

/* Create a 42-Megabyte partition: */
part_info.SLOT = 1;
part_info.TYPE = PMGR_PARTITION_HUGE_LBA;
part_info.START_SECTOR =  32;
part_info.LENGTH = 84432;

error_code = ioctl(pm_fd_ptr, IO_IOCTL_SET_PARTITION,
                     (uint32_t *) &part_info);
if ( error_code ) {
   printf("\nError creating partition %d!\n Error code: %d",
      1, error_code);
```

```
   _mqx_exit(1);
}/* Endif */

/* Create a 5-Megabyte partition: */
part_info.SLOT = 2;
part_info.TYPE = PMGR_PARTITION_FAT_12_BIT;
part_info.START_SECTOR = 84464;
part_info.LENGTH = 10000;

error_code = ioctl(pm_fd_ptr, IO_IOCTL_SET_PARTITION,
                     (uint32_t *) &part_info);

if ( error_code ) {
   printf("\nError creating partition %d!\n Error code: %d",
      2, error_code);
   _mqx_exit(1);
}/* Endif */
```

#### 3.8.2.4 IO_IOCTL_USE_PARTITION

This command directly sets partition parameters to use with the handle.

The third **ioctl()** parameter is the *PMGR_PART_INFO_STRUCT* pointer that is cast to *uint32_t **.

The information passed to this IOCTL call directly sets partition information associated with the handle without touching the underlying device. This provides with possibility to restrict access through the handle to certain part of the underlying device even for media without partition table in the first sector, i.e. the device may be partitioned in software.

Seek to the beginning of the just defined partition is performed when this IOCTL gets executed.

#### 3.8.2.5 IO_IOCTL_SEL_PART

This command selects partition to use with the handle.

The third **ioctl()** parameter points to *uint32_t* number which has to be between zero and four and represents the number of partition to select. If zero is specified no partition will be selected, i.e. whole device will be accessible through the handle.

It is not possible to directly select another partition if there is a partition already selected. Partition has to be deselected first, i.e. *IO_IOCTL_SEL_PART* has to be executed with pointer to zero as third parameter.

Seek to the beginning of the just selected partition or the device is performed when this IOCTL gets executed.

#### 3.8.2.6 IO_IOCTL_VAL_PART

This command validates partition table and checks partition type.

The third **ioctl()** parameter may be either *NULL* or pointer to *uint32_t* number which has to be between zero and four.

The IOCTL call checks partition table for validity. Then, it optionally checks type of partition whether it matches one of the FAT partition types. If the third parameter is pointer to zero only the partition table validity check is performed. If the third parameter is NULL, the type check is performed on a currently selected partition.

The IOCTL call with non-NULL third parameter is valid only if no partition is selected, i.e. the whole device is accessible through the handle.

If the partition type is checked and does not match any of the FAT partition types, *PMGR_UNKNOWN_PARTITION* is returned which indicates that the partition is valid but does not match any of the FAT types.

### 3.8.3 Return Codes for MFS

- *MFS_ACCESS_DENIED*
  - — Application attempted to modify a read-only file or a system file.
- *MFS_ALREADY_ASSIGNED*
- *MFS_ATTEMPT_TO_REMOVE_CURRENT_DIR*
- *MFS_BAD_DISK_UNIT*
  - — Operation on a file failed because that file is corrupted.
- *MFS_BAD_LFN_ENTRY*
  - — MFS failed to find a complete long file name within two clusters.
- *MFS_CANNOT_CREATE_DIRECTORY*
  - — MFS was unable to create the requested long directory name, usually because an invalid (illegal) directory name was specified.
- *MFS_DISK_FULL*
  - — Disk is full.
- *MFS_DISK_IS_WRITE_PROTECTED*
  - — Disk is write protected and could not be written to.
- *MFS_EOF*
  - — End of the file has been reached during a read. This is not a failure; it is only a warning.
- *MFS_ERROR_INVALID_DRIVE_HANDLE*
  - — The MFS *FILE_PTR* was invalid.
- *MFS_ERROR_INVALID_FILE_HANDLE*
  - — The MFS *FILE_PTR* was invalid.
- *MFS_ERROR_UNKNOWN_FS_VERSION*
  - — The drive contains an advanced FAT32 version. The MFS FAT32 version is not compatible. (There is currently only one FAT32 version, but this could change in the future.)
- *MFS_FAILED_TO_DELETE_LFN*
  - — MFS failed to completely delete a long file name. This results when MFS can not locate all of the long file name entries associated with a file.

- *MFS_FILE_EXISTS*
  - File already exists with the specified name.
- *MFS_FILE_NOT_FOUND*
  - File specified does not exist.
- *MFS_INSUFFICIENT_MEMORY*
  - MFS memory allocation failed. (MQX is out of memory or it has a corrupted memory pool.)
- *MFS_INVALID_CLUSTER_NUMBER*
  - A cluster number was detected that exceeds the maxumum number of clusters on the drive (or partition). This may be a result of a corrupted directory entry.
- *MFS_INVALID_DEVICE*
  - The underlying block mode driver does not support the block size command, or the block size is not legal (neither one of 512, 1024, 2048, or 4096 bytes).
- *MFS_INVALID_FUNCTION_CODE*
  - Not currently used.
- *MFS_INVALID_HANDLE*
  - One of the fields in a given *FILE_PTR* structure was invalid.
- *MFS_INVALID_LENGTH_IN_DISK_OPERATION*
  - Requested directory exceeds maximum in change-directory operation.
- *MFS_INVALID_MEMORY_BLOCK_ADDRESS*
  - *SEARCH_DATA_PTR* is NULL on find-first or fine-next file operation.
- *MFS_INVALID_PARAMETER*
  - One or more of the parameters passed to **_io_ioctl()** is invalid.
- *MFS_LOST_CHAIN*
  - This is not a critical error. It means there is a lost cluster chain which results in some wasted space. Operations on the drive continue normally.
- *MFS_NO_ERROR*
  - Function call was successful.
- *MFS_NOT_A_DOS_DISK*
  - Disk is not formatted at FAT12, FAT16, or FAT32 file system.
- *MFS_NOT_INITIALIZED*
  - Not currently returned.
- *MFSOPERATION_NOT_ALLOWED*
  - Returned when attempting a write operation when MFS is built in read-only mode, or a format operation when MFS is built without format functionality, or an attempt to rename a file to the same name.
- *MFS_PATH_NOT_FOUND*
  - Path name specified does not exist.
- *MFS_READ_FAULT*

— An error occurred reading from the disk.

- *MFS_ROOT_DIR_FULL*
  — Root directory on the drive has no more free entries for new files.
- *MFS_SECTOR_NOT_FOUND*
  — An error occurred while writing to the disk. The drive was formatted with incorrect parameters, or the partition table specified incorrect values.
- *MFS_SHARING_VIOLATION*
  — Produced by one of:
    – An attempt to close or format a drive that currently has files open.
    – An attempt to open a file to write that is already opened.
- *MFS_WRITE_FAULT*
  — An error occurred while writing to the disk.

### 3.8.4 Return Codes for the Partition Manager Device Driver

- *PMGR_INVALID_PARTITION*
  — The specified partition slot does not describe a valid partition.
- *PMGR_INSUF_MEMORY*
  — Attempt to allocate memory failed. MQX is out of memory or it has a corrupt memory pool.

### 3.8.5 Other Error Codes

An error was returned from the lower-level device driver.

# Chapter 4 Reference: Data Types

## 4.1 In This Chapter

Alphabetically sorted data-type descriptions for MFS.

## 4.2 _mfs_cache_policy

```
typedef enum {
   MFS_WRITE_THROUGH_CACHE=0,    // No write caching (only read caching)
   MFS_MIXED_MODE_CACHE=1,       // Write Caching allowed on file write only
   MFS_WRITE_BACK_CACHE=2        // Write Caching fully enabled
} _mfs_cache_policy;
```

## 4.3 MFS_DATE_TIME_PARAM

```
typedef struct mfs_date_time_param
{
   uint16_t * DATE_PTR;
   uint16_t * TIME_PTR;
} MFS_DATE_TIME_PARAM, * MFS_DATE_TIME_PARAM_PTR;
```

A pointer to the structure is used in *IO_IOCTL_GET_DATE_TIME* and *IO_IOCTL_SET_DATE_TIME* commands.

The first field is the *uint16_t* * to *uint16_t* variable in which the date is to be stored (for get) or read from (for set). The second field is the *uint16_t* * to *uint16_t* variable, in which the time is to be stored (for get) or read from (for set). See the **ioctl** description for details.

## 4.4 MFS_FILE_ATTR_PARAM

```
typedef struct mfs_file_attr_param
{
   char *   PATHNAME;
   /* Path name and filename of the file */
   unsigned char *  ATTRIBUTE_PTR;
   /* pointer to the attribute variable */
} MFS_FILE_ATTR_PARAM, * MFS_FILE_ATTR_PARAM_PTR;
```

A pointer to the structure is used in *IO_IOCTL_GET_FILE_ATTR* and *IO_IOCTL_SET_FILE_ATTR* commands.

The first field is the *char* * to the path name and filename of the file for which you want to get or set the attribute. The second field is the *unsigned char* * to the *char* variable in which the attribute is read from (for set), or in which the attribute is stored (for get).

## 4.5 MFS_GET_LFN_STRUCT

```
typedef struct mfs_get_lfn_struct
{
   char *   PATHNAME;
   /* Path name of the 8.3 name */
   char *   LONG_FILENAME;
   /* pointer to memory block in which to store the long name */
} MFS_GET_LFN_STRUCT, * MFS_GET_LFN_STRUCT_PTR;
```

A pointer to this structure is used in *IO_IOCTL_GET_LFN* commands.

The first field is the *char* * to the path name or file name of the file that we want to get the long file name of. The second field is the *char* * to pre-allocated space in which to store the long file name of the requested file.

## 4.6 MFS_IOCTL_FORMAT_PARAM

```
typedef struct mfs_ioctl_format
{
   MFS_FORMAT_DATA_PTR  FORMAT_PTR; /* Points to format data */
   uint32_t *          COUNT_PTR;  /* Count the bad clusters */
} MFS_IOCTL_FORMAT_PARAM, * MFS_IOCTL_FORMAT_PARAM_PTR;
```

A pointer to the structure is used in calls to *IO_IOCTL_FORMAT* and *IO_IOCTL_FORMAT_TEST* commands.

The first field is a pointer to the *MFS_FORMAT_DATA* structure, which is explained at the beginning of this document. The second field is used only for the *IO_IOCTL_FORMAT_TEST* command. It is a pointer to the *uint32_t* variable in which the count of bad clusters is stored.

## 4.7 MFS_RENAME_PARAM

```
typedef struct mfs_rename_param
{
   char *                OLD_PATHNAME;
   char *                NEW_PATHNAME;
} MFS_RENAME_PARAM, * MFS_RENAME_PARAM_PTR;
```

A pointer to the structure used in *IO_IOCTL_RENAME_FILE* commands.

The first field is the *char* * to a string that contains the path name and file name of the file to move or rename. The second field is the *char* * to the new path name or filename.

## 4.8 MFS_SEARCH_PARAM

```
typedef struct mfs_search_param
{
   unsigned char ATTRIBUTE;
   /* Attribute of the wanted file */
   char *            WILDCARD;
   /* Wildcard representation of the file */
   MFS_SEARCH_DATA_PTR     SEARCH_DATA_PTR;
   /* Points to search data */
} MFS_SEARCH_PARAM, * MFS_SEARCH_PARAM_PTR;
```

A pointer to the structure is used in *IO_IOCTL_FIND_FIRST_FILE* commands.

The first field is the `unsigned char` variable that contains the attribute of the file that you are searching for. The second field is the *char* * to a string containing the file name, including the path name, of the file that you are searching for. It can include wildcard characters. The third parameter is a pointer to the *MFS_SEARCH_DATA* structure. See the *IO_IOCTL_FIND_FIRST_FILE* command explanation for details.