CrossWorks Mass Storage Library

Version: 3.0

Contents

CrossWorks Mass Storage Library

CrossWorks Mass Storage Library

The *CrossWorks Mass Storage Library* is a collection of functions and device drivers that add mass storage capability to your application. We have primarily designed the Mass Storage Library to work well on reduced-memory real-time embedded systems that require mass storage, but you can equally well use the library on faster processors with more memory.

The Mass Storage Library is designed to run exclusively in the CrossWorks tasking environment; if your application doesn't use tasking and you wish to use this product then you must convert your application to run in a tasking environment which is simple enough to do. If you are using some other real time operating system, then using the Mass Storage Library is not viable and should seek a product that integrates well with your existing RTOS—or ditch that RTOS and use our excellent CTL tasking environment instead.

As you would expect, the Mass Storage Library integrates with other components in the CrossWorks Target Library. For instance, the Mass Storage Library uses the CrossWorks Device Library to provide physical-layer I/O to devices. The Mass Storage Library both integrates with the CrossWorks Streams framework.

Object Code Evaluation License

If you are evaluating the Mass Storage Library for use in your product, the following terms apply.

General terms

The source files and object code files in this package are not public domain and are not open source. They represent a substantial investment undertaken by Rowley Associates to assist CrossWorks customers in developing solutions using well-written, tested code.

Library Evaluation License

Rowley Associates grants you a license to the Object Code provided in this package solely to evaluate the performance and suitability of this library for inclusion into your products. You are prohibited from extracting, disassembling, and reverse engineering the Object Code in this package.

Object Code Commercial License

If you have paid to use the Mass Storage Library in your product, the following terms apply.

General terms

The source files and object code files in this package are not public domain and are not open source. They represent a substantial investment undertaken by Rowley Associates to assist CrossWorks customers in developing solutions using well-written, tested code.

Object Code Commercial License

If you hold a paid-for Object Code Commercial License for this product, you are free to incorporate the object code in your own products without royalties and without additional license fees. This Library is licensed to you PER DEVELOPER and is associated with a CrossWorks Product Key which, when combined, forms the entitlement to use this library. You must not provide the library to other developers to link against: each developer that links with this Library requires their own individual license.

<ctl_ms.h>

API Summary

Sector cache	
CTL_MS_DIRENTS_PER_SECTOR	There are 16 directory entries per sector.
CTL_MS_SECTOR_BUFFER_t	Representation of one 512-byte sector
CTL_MS_SECTOR_SIZE	There are 512 bytes per sector. We do not support media with
ctl_ms_borrow_sector_cache_memory	Borrow memory from sector cache
ctl_ms_flush_sector_cache	Write sector cache to media
ctl_ms_print_sector_cache	Print the contents of the sector cache
ctl_ms_purge_sector_cache	Purge all data from the sector cache
ctl_ms_return_sector_cache_memory	Return borrowed memory to sector cache
Errors	
CTL_MS_ERROR_t	Mass Storage library errors
File functions	
ctl_ms_close_file	Close an open file
ctl_ms_create_file	Create a file on a volume
ctl_ms_feof	End-of-file predicate
ctl_ms_flush_file	Flush all unwritten data
ctl_ms_fputc	Write a character to a file
ctl_ms_ftell	Read current file position
ctl_ms_get_attributes	Get the attributes of a file
ctl_ms_open_file	Open a file for reading or writing
ctl_ms_open_file_relative	Open a file for reading or writing in a folder
ctl_ms_read_block	Read a fixed-size block from a file
ctl_ms_read_char	Read a single character from a file
ctl_ms_read_string	Read a string from a file
ctl_ms_set_attributes	Set the attributes of a file
ctl_ms_write_block	Write a fixed-size block to a file
ctl_ms_write_string	Write a string to a file
Folder functions	
ctl_ms_change_current_folder	Change the working folder
ctl_ms_create_folder	Create a folder

ctl_ms_get_file_length	Get the length of a file
ctl_ms_remove_folder	Remove a folder
ctl_ms_rename_file	Rename a file
ctl_ms_set_file_length	Set the length of a file
Volume functions	
ctl_ms_get_volume_label	Get the volume label of a volume
ctl_ms_mount_volume	Mount a super-floppy or the default partition
ctl_ms_mount_volume_at_sector	Mount a volume or partition
ctl_ms_remove_file	Remove a file from a volume
ctl_ms_set_volume_label	Set the volume label of a volume
ctl_ms_unmount_volume	Unmount a volume
ctl_ms_unused_clusters	Calculate the number of unused clusters on a volume
Utility functions	
ctl_ms_decode_access_time	Read file last-access time from directory entry
ctl_ms_decode_attributes	Read attributes from directory entry
ctl_ms_decode_create_time	Read file creation time from directory entry
ctl_ms_decode_file_size	Read file size from directory entry
ctl_ms_decode_modify_time	Read file modification time from directory entry
ctl_ms_dos_to_timeval	Convert DOS time to timeval
ctl_ms_is_folder	
ctl_ms_read_sector	Read a sector direct from media
ctl_ms_timeval_to_dos	Convert timeval to DOS time
SD and MMC functions	
ctl_ms_read_cid	Read the card ID from media
ctl_ms_read_csd	Read the card-specific data from media
ctl_ms_read_scr	Read the SD Configuration Register from media
***** UNASSIGNED GROUP *****	
ctl_ms_sense_total_sectors	Determine total number of sectors a volume holds
Status functions	
ctl_ms_read_dirent	Get information on file or directory
Global functions	
ctl_ms_register_error_decoder	Register mass storage error decoder with runtime
Utility	
ctl_ms_update_working_directory	Change working directory

CTL_MS_DIRENTS_PER_SECTOR

Synopsis

```
#define CTL_MS_DIRENTS_PER_SECTOR  (CTL_MS_SECTOR_SIZE/CTL_MS_DIRENT_SIZE)
```

CTL_MS_ERROR_t

Synopsis

```
typedef enum {
  CTL_MS_NOT_OPEN_ERROR,
  CTL_MS_NAME_ERROR,
  CTL_MS_READONLY_ERROR,
  CTL_MS_SEEK_ERROR,
  CTL_MS_MODE_ERROR,
  CTL_MS_DISK_FULL_ERROR,
  CTL_MS_PATH_NOT_FOUND,
  CTL_MS_DIR_NOT_EMPTY_ERROR,
  CTL_MS_DIR_FULL_ERROR,
  CTL_MS_NOT_FOUND_ERROR,
  CTL_MS_IN_USE_ERROR,
  CTL_MS_ACCESS_ERROR,
  CTL_MS_EXISTS_ERROR,
  CTL_MS_BAD_FAT_ERROR,
  CTL_MS_READ_PAST_EOF_ERROR,
  CTL_MS_MEDIA_REMOVED_ERROR,
  CTL_MS_NO_FILESYSTEM_ERROR,
  CTL_MS_UNSUPPORTED_MEDIA_ERROR,
  CTL_MS_BAD_VOLUME_ERROR,
  CTL_MS_NOT_MOUNTED_ERROR,
  CTL_MS_CONFIGURATION_ERROR,
  CTL_MS_DELAYED_WRITE_ERROR,
  CTL_MS_CACHE_FULL,
  CTL_MS_INTERNAL_ERROR,
  CTL_MS_UNSUPPORTED_OPERATION,
  CTL_MS_MEDIA_LOCKED,
  CTL_MS_SD_ERROR
} CTL_MS_ERROR_t;
```

Description

CTL_MS_ERROR_t defines the errors reported by the Mass Storage Library.

CTL_MS_NO_FILESYSTEM_ERROR

No file system found on media. This this indicates that when attempting to mount the first file system on an MBR-partitioned disk, the MBR did not contain an active entry for any of the MBR partitions.

CTL_MS_UNSUPPORTED_MEDIA_ERROR

Media is not supported. This indicates that the volume, although it's a FAT volume, is not supported by the release of the mass storage library. Alternatively, it can be that the low-level media drivers have detected that the physical format of the media and its interface cannot be supported because of incompatibilities at the physical layer.

CTL_MS_BAD_VOLUME_ERROR

Volume is invalid. This indicates that the mass storage library detected an error in the format of the FAT volume when mounting it. This can indicate that the volume is simply not a FAT volume or that there is a more serious issue with the layout of the volume headers. If you can, mount the volume in a PC and check its integrity.

CTL_MS_NOT_MOUNTED_ERROR

File system is not mounted. This indicates that a file operation was requested on a volume that is not mounted.

CTL_MS_CONFIGURATION_ERROR

Library configuration error. This indicates that the library has been compiled incorrectly and the internal checks on data structure layout and sizes has failed. Please check the compilation options you provided that affect the compiler's data layout.

CTL_MS_DELAYED_WRITE_ERROR

Write error flushing sector cache. This indicates that the delayed write of a dirty sector to the media failed.

CTL_MS_CACHE_FULL

Sector cache is full. This indicates that the mass storage library required a sector to be read into the sector cache but all sectors in the sector cache are already locked which precludes reading the requested sector. This can happen if you open may files and simultaneously write to them without ensuring that the sector cache is created with at least one sector cache entry per open file.

CTL_MS_CACHE_FULL

Internal mass storage error. This this indicates that the mass storage library detected an error that should not happen. Even though the mass storage library is well-tested, there are internal checks in the library to ensure proper operation. If you receive this error, it could show a real error in the mass storage library, but more likely is an error in user code that has corrupted the data structures maintained by the mass storage library.

CTL_MS_UNSUPPORTED_OPERATION

Unsupported operation. This indicates that an operation was requested and is not appropriate, or cannot be honored, given the parameters supplied. For instance, requesting the CID from a volume that is not an SD or MMC card is not appropriate and results in this error.

CTL_MS_MEDIA_LOCKED

Media is locked. This this indicates that a volume cannot be ejected by **ctl_ms_unmount_volume** because a client has a sector on the volume locked for read or write.

CrossWorks Mass Storage Library

CTL_MS_SECTOR_BUFFER_t

Synopsis

```
typedef struct {
  unsigned char bytes[];
  unsigned short words[];
  unsigned long longs[];
  CTL_MS_DIRENT_t dirent[];
} CTL_MS_SECTOR_BUFFER_t;
```

CrossWorks Mass Storage Library

CTL_MS_SECTOR_SIZE

Synopsis

```
#define CTL_MS_SECTOR_SIZE        512
```

any other sector size.

ctl_ms_borrow_sector_cache_memory

Synopsis

```
void *ctl_ms_borrow_sector_cache_memory(int n);
```

Description

ctl_ms_borrow_sector_cache_memory borrows **n** contiguous sector cache entries and prevents **ctl_ms_read_lock_sector** and **ctl_ms_write_lock_sector** from using those buffers. **ctl_ms_borrow_sector_cache_memory** returns zero if **n** contiguous entries cannot be found.

You can borrow as much as you like from the sector cache but in doing so you may starve the file system of buffers that it requires to manage files and folders on the mounted volume. If the file system is starved, it will fail gracefully without damaging the volume.

ctl_ms_change_current_folder

Synopsis

```
CTL_STATUS_t ctl_ms_change_current_folder(const char *path);
```

Description

ctl_ms_change_current_folder changes the current working folder of the task to **path**.

Return Value

ctl_ms_change_current_folder returns a standard status code.

Thread Safety

ctl_ms_change_current_folder is thread-safe. However, note that the current folder for a volume is shared between *all* threads, so changing the current folder in one thread will affect the current folder of *all* threads.

ctl_ms_close_file

Synopsis

```
CTL_STATUS_t ctl_ms_close_file(CTL_STREAM_t f);
```

Description

ctl_ms_close_file closes the file **f**. All unwritten data is flushed to the physical media.

ctl_ms_close_file returns a standard status code.

Thread Safety

ctl_ms_close_file is thread-safe.

ctl_ms_create_file

Synopsis

```
CTL_STREAM_t ctl_ms_create_file(const char *name,
                                unsigned attrib);
```

Description

ctl_ms_create_file creates a file with name **name**. The file is created with the attributes **attrib**.

If the file is created without error, **f** is initialized and can be used for further file operations.

Return Value

ctl_ms_create_file returns a standard status code.

Thread Safety

this is thread-safe.

ctl_ms_create_folder

Synopsis

```
CTL_STATUS_t ctl_ms_create_folder(const char *path);
```

Description

ctl_ms_create_folder creates the folder with the name pointed to by **str**.

Return Value

ctl_ms_create_folder returns a standard status code.

Thread Safety

ctl_ms_create_folder is thread-safe.

ctl_ms_decode_access_time

Synopsis

```
void ctl_ms_decode_access_time(const CTL_MS_DIRENT_t *dirent,
                               timeval *tv);
```

Description

ctl_ms_decode_access_time extracts the last-access time of the file or directory from the directory entry pointed to by **dirent** into the **struct timeval** pointed to by **tv**.

ctl_ms_decode_attributes

Synopsis

```
unsigned ctl_ms_decode_attributes(const CTL_MS_DIRENT_t *dirent);
```

Description

ctl_ms_decode_attributes returns the attributes from the directory entry pointed to by **dirent**.

ctl_ms_decode_create_time

Synopsis

```
void ctl_ms_decode_create_time(const CTL_MS_DIRENT_t *dirent,
                               timeval *tv);
```

Description

ctl_ms_decode_create_time extracts the creation time of the file or directory from the directory entry pointed to by **dirent** into the **struct timeval** pointed to by **tv**.

ctl_ms_decode_file_size

Synopsis

```
unsigned long ctl_ms_decode_file_size(const CTL_MS_DIRENT_t *dirent);
```

Description

ctl_ms_decode_file_size returns the file size field of the directory entry pointed to by **dirent**. No file on a FAT file system can be larger than 2GB which fits into an `unsigned long`.

ctl_ms_decode_modify_time

Synopsis

```
void ctl_ms_decode_modify_time(const CTL_MS_DIRENT_t *dirent,
                               timeval *tv);
```

Description

ctl_ms_decode_modify_time extracts the last-modification time of the file or directory from the directory entry pointed to by **dirent** into the **struct timeval** pointed to by **tv**.

ctl_ms_dos_to_timeval

Synopsis

```
void ctl_ms_dos_to_timeval(timeval *tp,
                           unsigned short date,
                           unsigned short time,
                           unsigned short ms);
```

Description

ctl_ms_dos_to_timeval converts the DOS time used in FAT directory entries into a time pointed to by **tp**.

ctl_ms_feof

Synopsis

```
int ctl_ms_feof(CTL_STREAM_t s);
```

Description

ctl_ms_feof indicates whether the file **file** is positioned at the end of file.

ctl_ms_feof returns zero is **file** is not positioned at the end of file and non-zero (true) if it is.

Thread Safety

ctl_ms_feof is thread-safe.

ctl_ms_flush_file

Synopsis

```
CTL_STATUS_t ctl_ms_flush_file(CTL_STREAM_t s);
```

Description

ctl_ms_flush_file flushes all unwritten data of the file **s** to the media.

Return Value

ctl_ms_flush_file returns a standard status code.

Thread Safety

ctl_ms_flush_file is thread-safe.

ctl_ms_flush_sector_cache

Synopsis

```
CTL_STATUS_t ctl_ms_flush_sector_cache(void);
```

Description

ctl_ms_flush_sector_cache writes all dirty sectors to the storage media. The cache entries remain valid so they are immediately ready for a subsequent read request.

If you wish to invalidate the whole cache so that it is empty, use **ctl_ms_purge_sector_cache**.

All sectors are written to the media. If there is an error writing to the media for any sector, **ctl_ms_flush_sector_cache** returns **CTL_MS_DELAYED_WRITE_ERROR**.

See Also

ctl_ms_purge_sector_cache

ctl_ms_fputc

Synopsis

```
CTL_STATUS_t ctl_ms_fputc(CTL_STREAM_t s,
                          int ch);
```

Description

ctl_ms_fputc writes the character **ch** to the file **f**. The character is written without any translation which means that the C character, '\n', for instance, is not translated to a CR, LF sequence on output.

Return Value

ctl_ms_fputc returns a standard status code.

Thread Safety

ctl_ms_fputc is thread-safe.

ctl_ms_ftell

Synopsis

```
CTL_STATUS_t ctl_ms_ftell(CTL_STREAM_t s);
```

Description

ctl_ms_ftell returns the current position of the file **f**.

Thread Safety

ctl_ms_ftell is thread-safe.

ctl_ms_get_attributes

Synopsis

```
CTL_STATUS_t ctl_ms_get_attributes(const char *name,
                                   unsigned *attrib);
```

Description

ctl_ms_get_attributes gets the attributes of the file with name **name** on volume **vol** and writes them to the object **attrib**.

Return Value

ctl_ms_get_attributes returns a standard status code.

Thread Safety

ctl_ms_get_attributes is thread-safe.

ctl_ms_get_file_length

Synopsis

```
CTL_STATUS_t ctl_ms_get_file_length(CTL_STREAM_t s);
```

Description

ctl_ms_get_file_length gets the length of the open file **s** and returns it.

ctl_ms_get_volume_label

Synopsis

```
CTL_STATUS_t ctl_ms_get_volume_label(const char *vol,
                                     char *name);
```

Description

ctl_ms_get_volume_label reads the volume label of volume **vol** to the string pointed to by **name**. **name** must be able to hold at least 13 characters.

Return Value

ctl_ms_get_volume_label returns a standard status code.

Thread Safety

ctl_ms_get_volume_label is thread-safe.

ctl_ms_is_folder

Description

ctl_ms_is_folder inquires whether the path **str** designates a folder.

Return Value

ctl_ms_is_folder returns an extended status code: negative if there is an error accessing the path, 0 if the path does not designate a folder, and a positive value if the path does designate a folder.

ctl_ms_mount_volume

Synopsis

```
CTL_STATUS_t ctl_ms_mount_volume(const char *volume);
```

Description

ctl_ms_mount_volume mounts the volume **vol** using the FAT block driver **driver**. **vol** must be first initialized using **ctl_ms_init_volume**.

ctl_ms_mount_volume first reads boot sector (sector zero) of the volume and tries to determine if the volume is in super-floppy format or has a partition table. If the volume is in super-floppy format, it is mounted directly. If the volume has a partition map, the first valid partition on the drive is mounted.

If you need to mount a particular partition or a partition at a non-standard address, you can use **ctl_ms_mount_volume**.

Return Value

ctl_ms_mount_volume returns a standard status code.

Thread Safety

ctl_ms_mount_volume is thread-safe.

ctl_ms_mount_volume_at_sector

Synopsis

```
CTL_STATUS_t ctl_ms_mount_volume_at_sector(const char *volume,
                                           CTL_MS_LBA_t start_sector);
```

Description

ctl_ms_mount_volume_at_sector mounts the super-floppy volume without a master boot record or an MBR-partitioned volume on a disk. **start_sector** is the LBA of the first sector of the volume on the media for a super-floppy, or the LBA of the first sector of the partition to mount for a volume with an MBR.

Return Value

ctl_ms_mount_volume_at_sector returns a standard status code.

Thread Safety

ctl_ms_mount_volume_at_sector is thread-safe.

ctl_ms_open_file

Synopsis

```
CTL_STREAM_t ctl_ms_open_file(const char *name,
                              CTL_MS_MODE_t mode);
```

Description

ctl_ms_open_file opens the file name **name** on the volume **vol** for reading or writing according to the parameter **mode**.

Return Value

ctl_ms_open_file returns a standard status code.

Thread Safety

ctl_ms_open_file is thread-safe.

ctl_ms_open_file_relative

Synopsis

```
CTL_STREAM_t ctl_ms_open_file_relative(const char *base,
                                       const char *name,
                                       CTL_MS_MODE_t mode);
```

Description

ctl_ms_open_file_relative opens the file name **name** relative to the directory **base** on the volume **vol** for reading or writing according to the parameter **mode**.

The effect of this is to open the file whose path is the concatenation of **base**, a directory separator, and **name**.

Return Value

ctl_ms_open_file_relative returns a standard status code.

Thread Safety

ctl_ms_open_file_relative is thread-safe.

ctl_ms_print_sector_cache

Synopsis

```
void ctl_ms_print_sector_cache(CTL_STREAM_t s);
```

Description

ctl_ms_print_sector_cache prints the management data for the sector cache to the stream **s**.

Note that the sector cache mutex is locked when the sector cache is being printed and, hence, if you direct output to a file stream there is a possibility of deadlock if the file system requests a new sector buffer from the sector cache.

ctl_ms_purge_sector_cache

Synopsis

```
CTL_STATUS_t ctl_ms_purge_sector_cache(void);
```

Description

ctl_ms_purge_sector_cache writes all dirty sectors to the storage media and then invalidates all cache entries so that nothing remains in the cache. Before ejecting the media you should call **ctl_ms_purge_sector_cache** to ensure that all cached data is written and the physical storage media is consistent.

If you wish to only ensure that unwritten data is flushed such that the storage media is consistent, but allow the cache to remain valid, use **ctl_ms_flush_sector_cache**.

If there is an error writing to the media for any sector, **ctl_ms_purge_sector_cache** returns **CTL_MS_DELAYED_WRITE_ERROR**.

See Also

ctl_ms_flush_sector_cache

ctl_ms_read_block

Synopsis

```
CTL_STATUS_t ctl_ms_read_block(CTL_STREAM_t s,
                               void *data,
                               size_t len);
```

Description

ctl_ms_read_block reads bytes from the file **s** into the memory pointed to by **data**.

ctl_ms_read_block returns the number of bytes read or a CTL error code if an error occurred whilst reading.

Thread Safety

ctl_ms_read_block is thread-safe.

ctl_ms_read_char

Synopsis

```
CTL_STATUS_t ctl_ms_read_char(CTL_STREAM_t s);
```

Description

ctl_ms_read_char reads one character from the file **f**. Operating-system-specific end-of-line combinations are not translated to the C '\n' character; this must be done by the client.

ctl_ms_read_char returns a non-negative character if the character was read without error, otherwise otherwise an error code. Specifically, reading beyond the end of file returns the error **CTL_MS_READ_PAST_EOF_ERROR**.

Thread Safety

ctl_ms_read_char is thread-safe.

ctl_ms_read_cid

Synopsis

```
CTL_STATUS_t ctl_ms_read_cid(const char *volume,
                             unsigned char *cid);
```

Description

ctl_ms_read_cid reads the card ID for the the volume **vol**. The volume **vol** must refer to a mounted device that has an SD card or MMC card mounted. If **vol** is some other type of device, **ctl_ms_read_cid** returns an error.

ctl_ms_read_csd

Synopsis

```
CTL_STATUS_t ctl_ms_read_csd(const char *volume,
                             unsigned char *csd);
```

Description

ctl_ms_read_csd reads the card-specific data for the the volume **vol**. The volume **vol** must refer to a mounted device that has an SD card or MMC card mounted. If **vol** is some other type of device, **ctl_ms_read_csd** returns an error.

ctl_ms_read_dirent

Synopsis

```
CTL_STATUS_t ctl_ms_read_dirent(const char *path,
                                CTL_MS_DIRENT_t *dirent);
```

ctl_ms_read_scr

Synopsis

```
CTL_STATUS_t ctl_ms_read_scr(const char *volume,
                             unsigned char *scr);
```

Description

ctl_ms_read_scr reads the SCR from the media in volume **vol**. The volume **vol** must refer to a mounted device that has an SD card or MMC card mounted. If **vol** is some other type of device, **ctl_ms_read_scr** returns an error.

ctl_ms_read_sector

Synopsis

```
CTL_STATUS_t ctl_ms_read_sector(const char *volume,
                                CTL_MS_LBA_t lba,
                                CTL_MS_SECTOR_BUFFER_t **buf);
```

Description

ctl_ms_read_sector reads the sector with LBA **lba** on the volume **volume** into the sector cache and writes a buffer pointer to the sector into **buf**.

ctl_ms_read_string

Synopsis

```
CTL_STATUS_t ctl_ms_read_string(char *str,
                                size_t n,
                                CTL_STREAM_t file);
```

Description

ctl_ms_read_string reads a string from the file **file** into the string pointed to by **str.** The buffer for the string **str** is **n** characters long. **str** is terminated with a null character.

Return Value

ctl_ms_read_string returns a standard status code.

Thread Safety

ctl_ms_read_string is thread-safe.

ctl_ms_register_error_decoder

Synopsis

```
void ctl_ms_register_error_decoder(void);
```

Description

ctl_ms_register_error_decoder registers an error decoder with the CrossWorks runtime system such that **strerror** will correctly decode errors produced by the mass storage library.

ctl_ms_remove_file

Synopsis

```
CTL_STATUS_t ctl_ms_remove_file(const char *name);
```

Description

ctl_ms_remove_file removes the file **name** from the file system.

Return Value

ctl_ms_remove_file returns a standard status code.

Thread Safety

ctl_ms_remove_file is thread-safe.

ctl_ms_remove_folder

Synopsis

```
CTL_STATUS_t ctl_ms_remove_folder(const char *path);
```

Description

ctl_ms_remove_folder removes the folder with the name pointed to by **str**.

Return Value

ctl_ms_remove_folder returns a standard status code.

Thread Safety

ctl_ms_remove_folder is thread-safe.

ctl_ms_rename_file

Synopsis

```
CTL_STATUS_t ctl_ms_rename_file(const char *old_name,
                                const char *new_name);
```

Description

ctl_ms_rename_file renames the file with name **old_name** to **new_name**. **old_name** can be a full path name to a file, but **new_name** must only be a file name.

Return Value

ctl_ms_rename_file returns a standard status code.

Thread Safety

ctl_ms_rename_file is thread-safe.

ctl_ms_return_sector_cache_memory

Synopsis

```
CTL_STATUS_t ctl_ms_return_sector_cache_memory(void *addr,
                                               int n);
```

Description

ctl_ms_return_sector_cache_memory returns the previously-borrowed sector cache memory **addr** to the cache. **n** is the number of sectors that were borrowed.

ctl_ms_sense_total_sectors

Synopsis

```
CTL_STATUS_t ctl_ms_sense_total_sectors(const char *volume);
```

ctl_ms_sense_total_sectors senses the total number of sectors that the volume **volume** can hold. For MMC and SD cards, **ctl_ms_sense_total_sectors** reads the appropriate registers from the media and computes the total number of 512-byte sectors.

ctl_ms_sense_total_sectors will return an error code if the total number of sectors cannot be determined.

ctl_ms_set_attributes

Synopsis

```
CTL_STATUS_t ctl_ms_set_attributes(const char *name,
                                   unsigned attrib);
```

Description

ctl_ms_set_attributes sets the attributes of the file with name **name** on volume **vol** to **attrib**.

Return Value

ctl_ms_set_attributes returns a standard status code.

Thread Safety

ctl_ms_set_attributes is thread-safe.

ctl_ms_set_file_length

Synopsis

```
CTL_STATUS_t ctl_ms_set_file_length(CTL_STREAM_t s,
                                    unsigned long length);
```

Description

ctl_ms_set_file_length sets the length of the open file **s** to **length**. You cannot extend the file beyond its written length, but you can truncate it.

ctl_ms_set_volume_label

Synopsis

```
CTL_STATUS_t ctl_ms_set_volume_label(const char *path,
                                     const char *name);
```

Description

ctl_ms_set_volume_label sets the volume label of volume **vol** to **name**.

Return Value

ctl_ms_set_volume_label returns a standard status code.

Thread Safety

ctl_ms_set_volume_label is thread-safe.

ctl_ms_timeval_to_dos

Synopsis

```
void ctl_ms_timeval_to_dos(unsigned short *date,
                           unsigned short *time,
                           unsigned short *ms,
                           const timeval *tp);
```

Description

ctl_ms_timeval_to_dos converts the time pointed to by **tp** into 'DOS time' used in FAT directory entries.

ctl_ms_unmount_volume

Synopsis

```
CTL_STATUS_t ctl_ms_unmount_volume(const char *volume);
```

ctl_ms_unmount_volume unmounts the volume **vol**. Before the volume is unmounted, any dirty sectors in the sector cache are flushed to the media and then cleared.

If any file is open on the volume, **ctl_ms_unmount_volume** will fail with an error.

Return Value

ctl_ms_unmount_volume returns a standard status code.

Thread Safety

ctl_ms_unmount_volume is thread-safe.

ctl_ms_unused_clusters

Synopsis

```
CTL_STATUS_t ctl_ms_unused_clusters(const char *volume,
                                    CTL_MS_CLUSTER_t *unused);
```

Description

ctl_ms_unused_clusters computes the number of unused clusters on the volume named **volume**.

On volumes with large FAT tables, **ctl_ms_unused_clusters** may take a long time as the whole of the FAT is traversed to calculate the unused clusters.

Thread Safety

ctl_ms_unused_clusters is thread-safe. Note that although this function is thread-safe, it will lock out *all* file operations on the volume **vol** while computing the number of free clusters.

ctl_ms_update_working_directory

Synopsis

```
void ctl_ms_update_working_directory(char *path,
                                     size_t path_size,
                                     const char *dir);
```

Description

ctl_ms_update_working_directory changes the path **path**, which is a directory specification, by processing **dir**, which is a relative directory specification.

ctl_ms_write_block

Synopsis

```
CTL_STATUS_t ctl_ms_write_block(CTL_STREAM_t s,
                                const void *data,
                                size_t len);
```

Description

ctl_ms_write_block writes bytes from the memory pointed to by **data** to the file **s**.

ctl_ms_write_block returns the number of bytes written or a CTL error code if an error occurred whilst writing.

Thread Safety

ctl_ms_write_block is thread-safe.

ctl_ms_write_string

Synopsis

```
CTL_STATUS_t ctl_ms_write_string(CTL_STREAM_t f,
                                 const char *str);
```

Description

ctl_ms_write_string writes the string pointed to by **str** to the file **f**.

Return Value

ctl_ms_write_string returns a standard status code.

Thread Safety

ctl_ms_write_string is thread-safe.

<ctl_ms_low_level.h>

API Summary

Types	
CTL_MS_BLOCK_DRIVER_t	Mass storage low-level block driver
CTL_MS_VOLUME_t	Internal volume structure
Sector cache functions	
ctl_ms_flush_sectors_for_volume	Flush all sector cache entries for a volume
ctl_ms_invalidate_sector_cache_range	Invalidate a range of sectors
ctl_ms_invalidate_sector_cache_single	Invalidate a single sector
ctl_ms_read_lock_sector	Lock a sector into the cache for reading
ctl_ms_unlock_buffer	Unlock a sector buffer
ctl_ms_write_lock_sector	Lock a sector into the cache for writing

CTL_MS_BLOCK_DRIVER_t

Synopsis

```
typedef struct {
  CTL_MS_MEDIA_TYPE_t media;
  CTL_STATUS_t (*read_sectors)(void *, CTL_MS_LBA_t , void *, unsigned);
  CTL_STATUS_t (*write_sectors)(void *, CTL_MS_LBA_t , const void *, unsigned);
  CTL_STATUS_t (*init)(void *);
  CTL_STATUS_t (*fini)(void *);
} CTL_MS_BLOCK_DRIVER_t;
```

Description

CTL_MS_BLOCK_DRIVER_t contains the functions needed to read and write data on a volume.

Member	Description
media	The underlying technology for the media.
read_sectors	The method called to read multiple sectors from the media.
write_sectors	The method called to write multiple sectors to the media.

CTL_MS_VOLUME_t

Synopsis

```
typedef struct {
  CTL_MS_BLOCK_DRIVER_t *block_driver;
  CTL_MUTEX_t mutex;
  CTL_MS_VOLUME_STATE_t state;
  unsigned sectors_per_cluster;
  unsigned reserved_sector_count;
  unsigned root_entry_count;
  unsigned number_of_fats;
  CTL_MS_FILE_SYSTEM_FORMAT_t format;
  CTL_MS_LBA_t second_fat_offset;
  CTL_MS_LBA_t partition_addr;
  CTL_MS_LBA_t root_dir_sector;
  CTL_MS_LBA_t current_dir_sector;
  CTL_MS_LBA_t working_dir_sector;
  CTL_MS_LBA_t fat1_sector;
  CTL_MS_LBA_t fat2_sector;
  CTL_MS_LBA_t first_data_sector;
  CTL_MS_CLUSTER_t total_data_clusters;
  CTL_MS_CLUSTER_t free_cluster;
  CTL_MS_CLUSTER_t root_cluster;
  CTL_MS_LBA_t fsinfo_sector;
  CTL_STATUS_t error;
  char temp_name[];
  CTL_MS_FILE_tag *__open_files;
  CTL_MS_VOLUME_s *__next;
  const char *__name;
  const CTL_STREAM_DRIVER_t *methods;
  CTL_STATUS_t (*open_fcb)(CTL_MS_VOLUME_s *,
  CTL_MS_FCB_t *, const char *, const char *, CTL_MS_MODE_t);
} CTL_MS_VOLUME_t;
```

Description

CTL_MS_VOLUME_t describes the internal state of a volume which the mass storage library uses. It is not publicized by any function and all data inside it, if you wish to examine members, should be considered read-only. We do not guarantee that the structure will be stable across releases of the mass storage library.

Only the members **block_driver**, **mutex**, and **state** are valid for volumes that are not mounted; all other members should be considered invalid for offline volumes.

Member	Description
`block_driver`	This is public knowledge and you're responsible for populating it. Methods to read and write a single 512-byte sector.
`mutex`	Volume mutex; this is a per-volume mutex that is locked when the volume is accessed by the mass storage library.
`state`	The internal state of the volume. A volume can be in offline, online, and mounted states.

sectors_per_cluster	The number of 512-byte sectors per FAT cluster.
reserved_sector_count	The number of sectors that are marked as reserved at the start of the media.
root_entry_count	The number of directory entries in the root directory for FAT12 and FAT16 volumes.
number_of_fats	The number of FATs contained on the volume. This is either one or two.
fat_type	The type of FAT volume, either FAT12, FAT16, or FAT32.
second_fat_offset	The LBA offset from the start of the volume for the second FAT.
partition_addr	The LBA of the first sector of the volume on the media.
root_dir_sector	The LBA of the first sector of the volume's root directory.
current_dir_sector	The LBA of the internal current working directory. This does not correspond to the 'current directory' that operating systems such as MS-DOS, Windows, and Unix have. This is purely an internal convenience for the implementation of the mass storage library.
working_dir_sector	The LBA of the internal working directory. This is purely an internal convenience for the implementation of the mass storage library.
fat1_sector	The LBA of the first sector of the first FAT.
fat2_sector	The LBA of the first sector of the second FAT; if there is no second FAT, this member is undefined.
first_data_sector	The LBA of the first data sector relative to the start of the volume. The first data sector follows the reserved sectors, FATs, and root directory entries.
total_data_clusters	The number of data clusters for the volume.
free_cluster	The index of the first potentially-free cluster on the volume. The first two clusters of a FAT volume are reserved. This member is maintained internally by the mass storage library to accelerate finding free clusters in the FAT.
root_cluster	The cluster index of the root directory for FAT32 volumes. This member is undefined for FAT12 and FAT16 volumes.
fsinfo_sector	The LBA of the FSInfo sector relative to the start of the volume. This member is undefined for FAT12 and FAT16 volumes.

`error`	The last reported error for the volume. This is maintained internally by the mass storage library and has no relevance for any client.
`temp_name`	A temporary working name store for the mass storage library. This is maintained internally by the mass storage library and has no relevance for any client.
`open_files`	A linked list of files that are open on the volume.

ctl_ms_flush_sectors_for_volume

Synopsis

```
CTL_STATUS_t ctl_ms_flush_sectors_for_volume(CTL_MS_VOLUME_t *vol);
```

Description

ctl_ms_flush_sectors_for_volume flushes all sectors in the cache associated with the volume **vol** to the media. If a sector on the volume is locked for read or write, **ctl_ms_flush_sectors_for_volume** fails immediately, without writing any sector to the media, and the sector cache is unmodified.

ctl_ms_invalidate_sector_cache_range

Synopsis

```
void ctl_ms_invalidate_sector_cache_range(CTL_MS_VOLUME_t *vol,
                                          CTL_MS_LBA_t first,
                                          CTL_MS_LBA_t last);
```

Description

ctl_ms_invalidate_sector_cache_range invalidates the sectors **first** to **last** inclusive on volume **vol**. If the sectors are marked dirty, they are *not* written to the media.

You can use **ctl_ms_invalidate_sector_cache_range** function to notify the sector cache of changes that happen outside of its control. An example of this is if you write or modify the media directly using device driver functions without going through the sector cache functions **ctl_ms_read_lock_sector** or **ctl_ms_write_lock_sector**—because the media is written without the sector cache being aware of the changes you must invalidate all the sectors in the cache that you have changed or the media will become inconsistent and you may lose data on the volume.

ctl_ms_invalidate_sector_cache_single

Synopsis

```
void ctl_ms_invalidate_sector_cache_single(CTL_MS_VOLUME_t *vol,
                                           CTL_MS_LBA_t first);
```

Description

ctl_ms_invalidate_sector_cache_single invalidates a single sector **first** on volume **vol**. If the sectors is marked dirty, it is *not* written to the media.

See Also

ctl_ms_invalidate_sector_cache_range.

ctl_ms_read_lock_sector

Synopsis

```
CTL_MS_SECTOR_BUFFER_t *ctl_ms_read_lock_sector(CTL_MS_VOLUME_t *vol,
                                                CTL_MS_LBA_t sector);
```

Description

ctl_ms_read_lock_sector reads sector **sector** from volume **vol** into the sector cache if not already present. Sectors will be flushed to the media as necessary in order to make space in the cache for the requested sector. **ctl_ms_read_lock_sector** will return zero if there is an error reading the sector from the volume or if there was an error flushing a sector in order to make room for this one.

See Also

ctl_ms_unlock_buffer.

ctl_ms_unlock_buffer

Synopsis

```
CTL_STATUS_t ctl_ms_unlock_buffer(CTL_MS_SECTOR_BUFFER_t *buf);
```

Description

ctl_ms_unlock_buffer releases the sector buffer **buf** so that it can be flushed from the cache. Note that the buffer, if marked as dirty, is not immediately flushed to the media: you must call **ctl_ms_flush_sector_cache** in order to ensure that the contents of the media and the match the sector cache.

ctl_ms_write_lock_sector

Synopsis

```
CTL_MS_SECTOR_BUFFER_t *ctl_ms_write_lock_sector(CTL_MS_VOLUME_t *vol,
                                                 CTL_MS_LBA_t sector);
```

Description

ctl_ms_write_lock_sector reads sector **sector** from volume **vol** into the sector cache if not already present. Sectors will be flushed to the media as necessary in order to make space in the cache for the requested sector. The sector buffer associated with the sector will be marked dirty so that it will be flushed by **ctl_ms_flush_sector_cache** or when space is needed in the sector cache. **ctl_ms_write_lock_sector** will return zero if there is an error reading the sector from the volume.

See Also

ctl_ms_unlock_buffer.

<ctl_ms_private.h>

API Summary

***** UNASSIGNED GROUP *****	
CTL_MS_INVALID_CLUSTER	This is a private API for the core library. Please don't use this
ctl_ms_erase_cluster_chain	Erase a cluster chain and return it to the free list.
ctl_ms_start_enumeration	Start directory contents enumeration.
Private functions	
ctl_ms_check_volume_state	Checks the state of a volume
ctl_ms_cluster_to_sector	Converts a cluster number to a sector address
ctl_ms_find_fcb	Get file control block for stream
ctl_ms_lock_volume	Locks a volume for exclusive access
ctl_ms_read_fat_entry	Reads a cluster entry from the FAT
ctl_ms_sector_to_cluster	Converts a sector address to a cluster number
ctl_ms_unlock_volume	Unlocks a previously-locked volume

CTL_MS_INVALID_CLUSTER

because you'll only come to grief.

ctl_ms_check_volume_state

Synopsis

```
CTL_STATUS_t ctl_ms_check_volume_state(CTL_MS_VOLUME_t *vol);
```

ctl_ms_check_volume_state checks to see whether **vol** has been properly mounted. If the volume is properly mounted, **ctl_ms_check_volume_state** returns **CTL_NO_ERROR** otherwise **CTL_MS_NOT_MOUNTED_ERROR**.

ctl_ms_cluster_to_sector

Synopsis

```
CTL_MS_LBA_t ctl_ms_cluster_to_sector(const CTL_MS_VOLUME_t *vol,
                                      CTL_MS_CLUSTER_t n);
```

Description

ctl_ms_cluster_to_sector converts the cluster number **n** on volume **vol** to a sector address on the same volume.

See Also

ctl_ms_cluster_to_sector

ctl_ms_erase_cluster_chain

Synopsis

```
CTL_STATUS_t ctl_ms_erase_cluster_chain(CTL_MS_VOLUME_t *vol,
                                        CTL_MS_CLUSTER_t start_cluster);
```

Description

ctl_ms_erase_cluster_chain returns the cluster chain starting at **start_cluster** to the set of free clusters.

ctl_ms_find_fcb

Synopsis

```
CTL_MS_FCB_t *ctl_ms_find_fcb(CTL_STREAM_t s);
```

ctl_ms_lock_volume

Synopsis

```
void ctl_ms_lock_volume(CTL_MS_VOLUME_t *vol);
```

Description

ctl_ms_lock_volume locks the volume **vol** for exclusive access. The mutex associated with the volume is claimed and, as such, each call to **ctl_ms_lock_volume** to lock the volume must be paired with a call to **ctl_ms_unlock_volume**.

See Also

ctl_ms_unlock_volume

ctl_ms_read_fat_entry

Synopsis

```
CTL_MS_CLUSTER_t ctl_ms_read_fat_entry(CTL_MS_VOLUME_t *vol,
                                       CTL_MS_CLUSTER_t cluster);
```

Description

ctl_ms_read_fat_entry reads the FAT to find the next cluster after the cluster **cluster** in the linked list of clusters (the 'cluster' chain).

ctl_ms_sector_to_cluster

Synopsis

```
CTL_MS_CLUSTER_t ctl_ms_sector_to_cluster(const CTL_MS_VOLUME_t *vol,
                                          CTL_MS_LBA_t addr);
```

Description

ctl_ms_sector_to_cluster converts the sector address **addr** on volume **vol** to a cluster number on the same volume.

See Also

ctl_ms_sector_to_cluster

ctl_ms_start_enumeration

Synopsis

```
CTL_STATUS_t ctl_ms_start_enumeration(CTL_MS_VOLUME_t *vol,
                                      const char *filename,
                                      unsigned attrib,
                                      CTL_MS_DIR_ENUM_t *dir);
```

Description

ctl_ms_start_enumeration searches the volume **vol** in the folder starting at sector **sector** for entries that match **filename** and **attrib**.

You must provide a pointer to a directory entry structure, **dir**, to hold contextual information when enumerating the directory.

ctl_ms_unlock_volume

Synopsis

```
CTL_STATUS_t ctl_ms_unlock_volume(CTL_MS_VOLUME_t *vol);
```

Description

ctl_ms_unlock_volume unlocks the previously-locked volume **vol**. Each call to **ctl_ms_unlock_volume** must be paired with a call to **ctl_ms_lock_volume**.

See Also

ctl_ms_lock_volume

<ctl_ms_sd.h>

API Summary

SPI Device Drivers	
ctl_ms_sd_spi_read_sectors	Read sectors from a media card in SPI mode
ctl_ms_sd_spi_write_sectors	Write sectors from a media card in SPI mode
SPI	
ctl_ms_sd_spi_read_cid	Reads the CID register from a media card in SPI mode
ctl_ms_sd_spi_read_csd	Reads the CSD register from a media card in SPI mode
ctl_ms_sd_spi_read_scr	Reads the SCR register from a media card in SPI mode
ctl_ms_sd_spi_sense_media	Sense the media type (SD or MMC) attached to the drive

ctl_ms_sd_spi_read_cid

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_read_cid(CTL_MS_SD_DRIVER_t *driver,
                                    unsigned char *cid);
```

Description

ctl_ms_sd_spi_read_cid reads the card ID (CID) register from the media card into **cid** using the device driver **driver**. **cid** must point to a buffer that is at least 16 characters in size.

Thread Safety

ctl_ms_sd_spi_read_cid is thread-safe.

ctl_ms_sd_spi_read_csd

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_read_csd(CTL_MS_SD_DRIVER_t *driver,
                                    unsigned char *csd);
```

Description

ctl_ms_sd_spi_read_csd reads the card-specific data (CSD) register from the media card into **csd** using the device driver **driver**. **csd** must point to a buffer that is at least 16 bytes in size.

Thread Safety

ctl_ms_sd_spi_read_csd is thread-safe.

ctl_ms_sd_spi_read_scr

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_read_scr(CTL_MS_SD_DRIVER_t *driver,
                                    unsigned char *scr);
```

Description

ctl_ms_sd_spi_read_scr reads the system control register (SCR) from the media card into **scr** using the device driver **driver**. **scr** must point to a buffer that is at least 8 bytes in size.

Thread Safety

ctl_ms_sd_spi_read_scr is thread-safe.

ctl_ms_sd_spi_read_sectors

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_read_sectors(void *driver,
                                        CTL_MS_LBA_t addr,
                                        void *buf,
                                        unsigned count);
```

Description

ctl_ms_sd_spi_read_sectors reads **count** sectors at sector address **addr** into memory at **buf** using the SPI driver **driver**.

Thread Safety

ctl_ms_sd_spi_read_sectors is thread-safe.

ctl_ms_sd_spi_sense_media

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_sense_media(CTL_MS_SD_DRIVER_t *driver,
                                       int sense);
```

ctl_ms_sd_spi_write_sectors

Synopsis

```
CTL_STATUS_t ctl_ms_sd_spi_write_sectors(void *driver,
                                         CTL_MS_LBA_t addr,
                                         const void *buf,
                                         unsigned count);
```

Description

ctl_ms_sd_spi_write_sectors reads **count** sectors at sector address **addr** into memory at **buf** using the SPI driver **driver**.

Thread Safety

ctl_ms_sd_spi_write_sectors is thread-safe.