QNX® Aviage Multimedia Suite 1.2.0

MME API Library Reference

For QNX® Neutrino® 6.4.x

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published March 13, 2009.

Contents

About this Reference

The *MME API Library Reference* accompanies the QNX Aviage multimedia suite, release 1.1.0. It is intended for application developers who use the suite's MultiMedia Engine (MME) to develop multimedia applications.
This table may help you find what you need in the *MME API Library Reference*:

When you want to:	Go to:
Learn about MME API functions, data structures, enumerated types and constants.	MME API
Learn about MME events and the data structures they use.	MME Events
Learn about MME synchronization events, and synchronization error events.	MME Synchronization Events
Learn about MME playback events, and playback error events.	MME Playback Events
Learn about MME media copy and ripping events, and copy and ripping error events.	MME Media Copy and Ripping Events
Learn about MME metadata events.	MME Metadata Events
Learn about the MME database schema.	MME Database Schema Reference

Other MME documentation available to application developers includes:

Book	Description
Introduction to the MME	MME Architecture, Quickstart Guide, and FAQs.
MME Developer's Guide	How to use the MME to program client applications.
MME Utilities	Utilities used by the MME.
MME Configuration Guide	How to configure the MME.
MME Technotes	MME technical notes.
QDB Developer's Guide	QDB database engine programming guide and API library reference.

Note that the MME is a component of the QNX Aviage multimedia core package, which is available in the QNX Aviage multimedia suite of products. The MME is the main component of this core package. It is used for configuration and control of your multimedia applications.

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications. The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	`something you type`
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	`unsigned short`
Programming literals	`0xFF`, `"message string"`
Variable names	*stdin*
User-interface components	**Cancel**

We use an arrow (→) in directions for accessing menu items, like this:

> You'll find the **Other...** menu item under **Perspective→Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

WARNING: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (`/`) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support options

To obtain technical support for any QNX product, visit the **Support + Services** area on our website (`www.qnx.com`). You'll find a wide range of support options, including community forums.

Chapter 1

MME API

In this chapter. . .

This chapter describes publicly visible MME API:

- functions
- data structures
- enumerated types

Data structures and enumerated types that are used by only one API function are documented with the relevant functions. Event structures, enumerated types and constants are described in the chapter MME Events. If you do not find a structure, enumerated type or constant in the list below, refer to the index. Configuration constants are described in the *MME Configuration Guide*.

Headers and libraries

For the location of MME libraries and header files, see the section "Headers and libraries" in the *Release Notes* for your MME release.

Compiling client applications

The MME requires that client applications be compiled with FILE_OFFSET_BITS set to 64. For example:

```
# qcc -Amy_library [other_options] -DFILE_OFFSET_BITS=64
```

For more information about compiling client applications for the MME, see **QCC, qcc** in the *QNX Neutrino Utilities Reference*.

Alphabetical list of MME functions, data structures, enumerated types and constants

FTYPE_*
METADATA_*
mm_audio_format_t
mm_audio_lang_ext
mm_audio_type
mm_bitrate_t
mm_blocked_uops
mm_display_mode
mm_dvd_status_t
mm_media_status_t
mm_metadata_t
mm_subpict_lang_ext
mm_uop_t
mm_video_angle_info_t
mm_video_audio_info_t

mm_video_info_t
mm_video_properties_t
mm_video_status_t
mm_video_subtitle_info_t
mme_audio_get_status()
mme_bookmark_create()
mme_bookmark_delete()
mme_buffer_status_t
mme_button()
mme_charconvert_setup()
mme_connect()
mme_copy_info_t
mme_delete_mediastores()
mme_device_get_config()
mme_device_set_config()
mme_directed_sync_cancel()
mme_disconnect()
mme_dvd_get_disc_region()
mme_dvd_get_status()
mme_explore_end()
mme_explore_hdl_t
mme_explore_info_free()
mme_explore_info_get()
mme_explore_info_t
mme_explore_playlist_find_file()
mme_explore_position_set()
mme_explore_size_get()
mme_explore_start()
MME_FORMAT_*
mme_get_api_timeout_remaining()
mme_getautopause()
mme_getccid()
mme_getclientcount()
mme_get_event()
mme_getlocale()
mme_get_logging()
mme_getrandom()
mme_getrepeat()
mme_getscanmode()
mme_get_title_chapter()
mme_hdl_t
mme_lib_column_set()
mme_media_get_def_lang()
mme_media_set_def_lang()
mme_mediacopier_add()
mme_mediacopier_add_with_metadata()

mme_mediacopier_cleanup()
mme_mediacopier_clear()
mme_mediacopier_disable()
mme_mediacopier_enable()
mme_mediacopier_get_mode()
mme_mediacopier_get_status()
mme_mediacopier_info_t
mme_mediacopier_remove()
mme_mediacopier_set_mode()
mme_metadata_alloc()
mme_metadata_create_session()
mme_metadata_extract_data()
mme_metadata_extract_string()
mme_metadata_extract_unsigned()
mme_metadata_free_session()
mme_metadata_getinfo_current()
mme_metadata_getinfo_file()
mme_metadata_getinfo_library()
mme_metadata_hdl_t
mme_metadata_image_cache_clear()
mme_metadata_image_load()
mme_metadata_image_unload()
mme_metadata_image_url_t
mme_metadata_info_t
mme_metadata_session_t
mme_metadata_set()
mme_mode_random_t
mme_mode_repeat_t
MME_MSCAP_*
mme_ms_clear_accurate()
mme_ms_metadata_done()
mme_ms_metadata_get()
mme_ms_restart()
mme_ms_state_t
mme_ms_statechange_t
mme_newtrksession()
mme_next()
mme_output_attr_t
mme_output_set_permanent()
mme_outputtype_t
mme_play()
mme_play_attach_output()
mme_play_bookmark()
mme_play_detach_output()
mme_play_file()
mme_play_get_info()

mme_play_get_output_attr()
mme_play_get_speed()
mme_play_get_status()
mme_play_get_zone()
mme_play_info_t
mme_play_offset()
mme_play_resume_msid()
mme_play_set_output_attr()
mme_play_set_speed()
mme_play_set_zone()
MME_PLAYLIST_*
mme_playlist_close()
mme_playlist_create()
mme_playlist_delete()
mme_playlist_generate_similar()
mme_playlist_hdl_t
mme_playlist_item_get()
mme_playlist_items_count_get()
mme_playlist_open()
mme_playlist_position_set()
mme_playlist_set_statement()
mme_playlist_sync()
mme_playstate_t
mme_playstate_speed_t
mme_play_status_t
mme_prev()
mme_register_for_events()
mme_resync_mediastore()
mme_rmtrksession()
mme_seek_title_chapter()
mme_seektotime()
mme_set_api_timeout()
mme_setautopause()
mme_set_debug()
mme_set_files_permanent()
mme_setlocale()
mme_set_logging()
mme_set_msid_resume_trksession()
mme_set_notification_interval()
mme_setpriorityfolder()
mme_setrandom()
mme_setrepeat()
mme_setscanmode()
mme_settrksession()
mme_shutdown()
MME_SLOTTYPE_*

mme_start_device_detection()
mme_stop()
MME_STORAGETYPE_*
mme_sync_cancel()
mme_sync_db_check()
mme_sync_directed()
mme_sync_file()
mme_sync_get_msid_status()
mme_sync_get_status()
MME_SYNC_OPTION_*
mme_sync_set_debug()
mme_sync_status_t
mme_time_t
mme_trksession_append_files()
mme_trksession_clear_files()
mme_trksession_get_info()
mme_trksession_resume_state()
mme_trksession_save_state()
mme_trksession_set_files()
mme_trksessionview_get_current()
mme_trksessionview_get_info()
mme_trksessionview_info_t
mme_trksessionview_metadata_get()
mme_trksessionview_readx()
mme_trksessionview_update()
mme_trksessionview_writedb()
mme_trksessionview_update()
mme_video_get_angle_info()
mme_video_get_audio_info()
mme_video_get_info()
mme_video_get_status()
mme_video_get_subtitle_info()
mme_video_set_angle()
mme_video_set_audio()
mme_video_set_properties()
mme_video_set_subtitle()
mme_zone_create()
mme_zone_delete()

Synopsis:

```
#include <mme/interface.h>

#define FTYPE_*
```

Description:

The constants FTYPE* define the media types the MME recognizes. The values listed in the table below are used by the *ftype* field in the:

- **`mme_play_info_t`** data structure
- **`library`** table
- **`nowplaying`** table

Constant	Value	Description
FTYPE_UNKNOWN	0	Unknown media type.
FTYPE_AUDIO	1	The media has audio only.
FTYPE_VIDEO	2	The media has video only.
FTYPE_AUDIOVIDEO	3	The media has both audio and video.
FTYPE_PHOTO	4	The media has images (photos).
FTYPE_DEVICE	5	The media can be accessed and played as one file. For example, play an entire DVD video rather than tracks on the DVD, or play streamed media.

Maintaining the accuracy of *ftype* fields

For some files, the file type cannot always be correctly established based only on the file extension (hence during the first synchronization pass). To ensure correct entries in the *ftype* field in the MME tables, the MME updates this field when it performs:

- the first synchronization pass
- the second synchronization pass
- normal playback, upon receiving the metadata update from **`io-media`**, if the MME is configured to *not* update the **`library`** from the **`nowplaying`** table (**`<UpdateLibraryFromNowplaying enabled="off"/>`**), the MME updates the *ftype* field in the **`library`** table *only*
- a mediacopier update of metadata, if the mediacopier is configured to make the metadata accurate before ripping (**`<UpdateMetadata enabled="true"/>`**)

Classification:

QNX Multimedia

See also:

MME_FORMAT_*, MME_MSCAP_*, MME_STORAGETYPE_*, MME_SYNC_OPTION_*, **`mediastores`** table in the appendix: MME Database Schema Reference

Definitions for metadata string types

Synopsis:

```
#include <mme/metadata.h>

#define METADATA_*
```

Description:

The constants METADATA_* define the metadata types for the strings used by MME functions that retrieve metadata for specific files: *mme_explore_info_get()* and *mme_ms_metadata_get()*. For information about how to compose the strings, see the chapter Metadata and Album Art in the *MME Developer's Guide*.

The table below lists current metadata types. All are types are of METADATA_FORMAT_*, as listed.

Constant	Format	Value	Description
METADATA_TITLE	STRING	"title"	The track title.
METADATA_ALBUM	STRING	"album"	The album with the track.
METADATA_ARTIST	STRING	"artist"	The track's artist.
METADATA_GENRE	STRING	"genre"	The track's genre.
METADATA_COMPOSER	STRING	"composer"	The track composer.
METADATA_PUBLISHER	STRING	"publisher"	The track publisher.
METADATA_NAME	STRING	"name"	The folder name. See METADATA_NAME below.

continued...

Constant	Format	Value	Description
METADATA_RELEASE_DATE	TM	"release_date"	The track's release date.
METADATA_YEAR	UNSIGNED	"year"	The track's release year.
METADATA_DURATION	UNSIGNED	"duration"	The duration of the track, in milliseconds.
METADATA_COMMENT	STRING	"comment or description"	A description of the track.
METADATA_TRACK_NUMBER	UNSIGNED	"track_number"	The track number.
METADATA_YEAR	UNSIGNED	"year"	The track's release year.
METADATA_COMMENT	STRING	"comment or description"	A description of the track.
METADATA_TRACK_NUMBER	UNSIGNED	"track_number"	The track number.

METADATA_NAME

The metadata for METADATA_NAME varies according to the context. With iPods, the name of a folder changes according to its parent folder. For example, the tracks from the album *Transparente* by Mariza, appear to be in different folders, depending on how the user arrives at the tracks:

- If the user is exploring the iPod through the **`artists`** folder, the value for METADATA_NAME is "Mariza", the name of the artist.
- If the user is exploring the iPod through the **`albums`**, the value for METADATA_NAME is "Transparente", the name of the album.

io-fs-media -dipod must be set to **short** for the MME to be able to retrieve metadata for tracks on an iPod.

METADATA_FORMAT_*

```
enum {
METADATA_FORMAT_INVALID = 0,
METADATA_FORMAT_DATA,
METADATA_FORMAT_STRING,
METADATA_FORMAT_TM,
METADATA_FORMAT_UNSIGNED,
};
```

The enumerated values METADATA_FORMAT_* describe the data types for metadata presentation, as follows:

- METADATA_FORMAT_INVALID — 0 (zero): invalid format.
- METADATA_FORMAT_DATA — blob.
- METADATA_FORMAT_STRING — character string.
- METADATA_FORMAT_TM — time.
- METADATA_FORMAT_UNSIGNED — unsigned integer.

Classification:

QNX Multimedia

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **mme_metadata_image_url_t**, **mme_metadata_session_t**

Synopsis:

```
#include <mm/types.h>

typedef struct mm_audio_format {
    char        codec[MM_CODEC_NAME_MAX_LEN];
    uint32_t    bitrate;
    uint32_t    samplerate;
    uint8_t     channels;
    uint8_t     bitrate_type;
    uint8_t     channel_type;
    uint8_t     reserve1;
    int32_t     reserve2;
    int32_t     reserve3;
} mm_audio_format_t;
```

Description:

The structure **mm_audio_format_t** provides information about the current state of an audio stream. It includes at least the members described in the table below.

Member	Type	Description
codec	**char**	Name of the audio codec. This member is the character string with the name of the audio codec. See “Audio codec” below.
bitrate	**uint32_t**	Average bitrate for the audio track, in bits per second.
samplerate	**uint32_t**	Sample bitrate, in hertz.
channels	**uint8_t**	Channel type. See Audio channels.
bitrate_type	**uint8_t**	Bitrate type. See **mm_bitrate_t** in this reference.
channel_type	**uint8_t**	Deprecated in MME 1.1.0. Do not use.
reserve1	**uint8_t**	For future use.
reserve2, 3	**int32_t**	For future use.

Audio codec

The MME API function *mme_audio_get_status()* uses the data structure **mm_audio_format_t**. The MME API function *mme_video_get_status()* uses the data structure **mm_video_info_t**. Both these structures include a member *codec*. The codec members of the structures **mm_video_info_t** and **mm_audio_format_t** hold character strings identifying the codec format for the video or audio. These strings can have a length of up to the number of bytes defined by MM_CODEC_NAME_MAX_LEN, which is currently 32 bytes.

Client applications can pass these character strings up to the end users to inform them of the codec format used by a video or audio track.

Audio channels

The *channels* member of the structure **mm_audio_format_t** describes the number of channels available in the audio stream. It can be set to any number defined as valid by the audio stream specification.

Example audio stream channels

Channels	Audio stream
1	mono
2	stereo
6	Dolby digital 5.1
6	DTS
8	Dolby digital 7.1
8	DTS_ES

Classification:

QNX Multimedia

See also:

mm_bitrate_t, **mme_video_audio_info_t**, **mm_video_audio_info_t**, **mm_video_info_t**, **mme_audio_get_status**, *video_get_status()*

Synopsis:

```
#include <mm/types.h>

enum mm_audio_lang_ext;
```

Description:

The enumerated type **mm_audio_lang_ext** defines video caption settings. Its values include:

- MM_CAPTIONS_NORMAL —normal captions.
- MM_VISUAL_IMPARED_AUDIO — captions for the visually impaired.
- MM_DIRECTORS_COMMENTS1 — director's comments.
- MM_DIRECTORS_COMMENTS2 — director's comments.

Classification:

QNX Multimedia

See also:

mm_video_info_t

Deprecated in MME 1.1.0. Do not use.

Synopsis:

```
#include <mm/types.h>

enum mm_audio_type;
```

Description:

mm_audio_types

The enumerated type **mm_audio_type** defines video audio types. Its values include:

- DOLBY_AC3
- LINEAR_PCM
- MPEG_1_2
- MPEG_2_EXT
- DTS
- SDDS
- MONO
- STEREO
- JOINT_STEREO
- DUAL_CHANNEL
- OTHER (255)

Classification:

QNX Multimedia

See also:

mm_audio_format_t, **mm_video_info_t**, **mme_audio_get_status**, *mme_video_get_status()*

Synopsis:

```
#include <mm/types.h>

enum mm_bitrate_t;
```

Description:

The enumerated type **mm_bitrate_t** defines streaming bitrate values. These values are listed below:

- MM_BITRATE_TYPE_UNKNOWN — unknown bit rate.
- MM_BITRATE_TYPE_CONSTANT — constant bitrate: the listed bitrate is always accurate.
- MM_BITRATE_TYPE_VARIABLE — variable bitrate: the bitrate of encoded packets is variable.

At present, all **io-media** graphs set **mm_bitrate_t** to MM_BITRATE_TYPE_UNKNOWN.

Classification:

QNX Multimedia

See also:

mm_audio_format_t, **mm_video_info_t**, **mme_audio_get_status**, *mme_video_get_status()*

Synopsis:

```
#include <mm/types.h>

enum mm_blocked_uops;
```

Description:

The enumerated type **mm_blocked_uops** defines values for the User Operations Prohibitions (UOP) bit mask. Its values and the behaviors they define are described below:

- UOP_BLOCK_NONE — no user prohibitions.
- UOP_BLOCK_TIME_PLAY_SEARCH — prohibit search to time.
- UOP_BLOCK_PTT_PLAY_SEARCH — prohibit search to chapters.
- UOP_BLOCK_TITLE_PLAY — prohibit play by title.
- UOP_BLOCK_STOP — prohibit stopping of video.
- UOP_BLOCK_GO_UP — prohibit “up” command.
- UOP_BLOCK_PREV_TOP_PG_SEARCH — prohibit
- UOP_BLOCK_NEXT_PG_SEARCH — prohibit search for next page.
- UOP_BLOCK_FORWARD_SCAN — prohibit forward scans.
- UOP_BLOCK_BACKWARD_SCAN — prohibit backward scans.
- UOP_BLOCK_MENU_CALL_TITLE — prohibit use of title menu.
- UOP_BLOCK_MENU_CALL_ROOT — prohibit use of root menu.
- UOP_BLOCK_MENU_CALL_SUB_PICTURE — prohibit use of sub-picture (subtitles) menu.
- UOP_BLOCK_MENU_CALL_AUDIO — prohibit changes to audio
- UOP_BLOCK_MENU_CALL_ANGLE — prohibit changes to angle.
- UOP_BLOCK_MENU_CALL_PTT — prohibit calls to chapter menu.
- UOP_BLOCK_RESUME — prohibit resume functionality.
- UOP_BLOCK_BUTTON — prohibit button functionality.
- UOP_BLOCK_STILL_OFF — prohibit turning off of stills.
- UOP_BLOCK_PAUSE_ON — prohibit pause.

- UOP_BLOCK_AUDIO_CHANGE — prohibit changes to audio properties.
- UOP_BLOCK_SUB_PICTURE_CHANGE — prohibit changes to sub-picture (subtitles).
- UOP_BLOCK_ANGLE_CHANGE — prohibit changes to video angle.
- UOP_BLOCK_KARAOKE_CHANGE — prohibit changes to karaoke settings.
- UOP_BLOCK_VIDEO_CHANGE — prohibit changes to video properties.

Classification:

QNX Multimedia

See also:

mm_dvd_status_t

Synopsis:

```
#include <mm/types.h>

enum display_mode;
```

Description:

The enumerated type **mm_display_mode** defines how a video is displayed. Its values and the behaviors they define are described below:

- MM_DISPLAY_MODE_NORMAL — fit the display: the picture is full screen.
- MM_DISPLAY_MODE_LETTERBOX — fit one dimension of the display and add black bars for other dimension: the picture is partial screen.
- MM_DISPLAY_MODE_PANSCAN — fit one dimension of the display and crop the other dimension: the picture is full screen.
- MM_DISPLAY_MODE_OPEN_MATTE — display full frame: the original content cropping is changed.

Classification:

QNX Multimedia

See also:

mm_video_properties_t, **mm_video_info_t**

Synopsis:

```
#include <mme/types.h>

typedef struct mm_dvd_status {
    struct mm_dvd_blocked {
        uint32_t    uop_mask;
        uint32_t    audio_mask;
        uint32_t    subpicture_mask;
    } blocked;
    uint32_t    domain;
    uint32_t    title;
    uint32_t    chapter;
    uint64_t    chapter_start_time;
    uint32_t    num_audio_streams;
    uint32_t    audio_stream;
    uint32_t    num_subtitle_streams;
    uint32_t    subtitle_stream;
    uint32_t    num_angles;
    uint32_t    angle;
    uint32_t    playback_pml;
    uint32_t    spare[4];
} mm_dvd_status_t;
```

Description:

The structure **mm_dvd_status_t** carries information about a DVD, including blocked functionality. It includes at least the members described in the table below.

Member	Type	Description
blocked	**struct**	Masks for User Operation Prohibitions. See **mm_dvd_blocked** below.
domain	**uint32_t**	The domain of the DVD.
title	**uint32_t**	The currently playing DVD title.
chapter	**uin32_t**	The currently playing chapter in the DVD title.
chapter_start_time	**uin64_t**	The offset (in milliseconds) of the chapter start from the start of the title.
num_audio_streams	**uin32_t**	The number of available audio streams.
audio_stream	**uin32_t**	The current audio stream.
num_subtitle_streams	**uin32_t**	The number of subtitle streams.

continued...

Member	Type	Description
subtitle_stream	**uin32_t**	The current subtitle stream.
num_angles	**uin32_t**	The number of angles.
angle	**uin32_t**	The current angle.
playback_pml	**uin32_t**	The parental management level needed for playback; set to 0 if no change in level is required.
spare	**uin32_t**	Spare.

mm_dvd_blocked

The structure **mm_dvd_blocked** contains masks indicating which User Operation Prohibitions (UOP), audio, and subpicture functionality is blocked for the current track. The UOP mask has bits set to indicate which DVD remote button operations are prohibited for the current track. The structure **mm_dvd_blocked** includes at least the members described in the table below.

Member	Type	Description
uop_mask	**uint32_t**	The bit mask for (UOP) User Operation Prohibitions. See **mm_blocked_uops** in this reference.
audio_mask	**uint32_t**	The mask indicating the audio functionality permissions set for the current track.
subpicture_mask	**uint32_t**	The mask indicating the subpicture functionality set for the current track.

mm_dvd_status_event_t

```
typedef struct mm_dvd_status_event {
    mm_dvd_status_t         status;
    mm_dvd_status_reason_t  reason;
} mm_dvd_status_event_t;
```

The structure **mm_dvd_status_event_t** carries information about a DVD, including its status, in **mm_dvd_status_t**, and the reason for the status event delivery, in **mm_dvd_status_reason_t**. It includes at least the members described in the table below.

Member	Type	Description
status	**struct**	Information about a DVD, including blocked functionality.
reason	**enum**	The reason for the DVD event delivery.

mm_dvd_status_reason_t

The enumerated type **mm_dvd_status_reason_t** is used to indicate the reason for which a DVD status update is delivered. It can be set to the following values:

- MM_DVD_DOMAIN_UPDATE — the DVD domain has changed.
- MM_DVD_TITLE_UPDATE — the DVD title has changed.
- MM_DVD_CHAPTER_UPDATE — th DVD chapter has changed.
- MM_DVD_ANGLE_UPDATE — the DVD angle has changed.
- MM_DVD_AUDIO_UPDATE — the DVD audio stream has changed.
- MM_DVD_SUBTITLE_UPDATE — the DVD subtitle stream has changed.
- MM_DVD_BLOCKED_UPDATE — the DVD user prohibitions have changed.
- MM_DVD_MENU_ACTIVE_UPDATE — the DVD active menu has changed.
- MM_DVD_PML_UPDATE — The parental management level is insufficient for playback, see *playback_pml* in **mm_dvd_status_t** for the needed level.

mm_dvd_domain

The enumerated type **mm_dvd_domain** is used to indicate the domain of the current track. The DVD specification defines four domains to which data can belong. **mm_dvd_domain** can be set to the following values:

- MM_DOMAIN_STOP — DVD is not playing.
- MM_DOMAIN_FP — First Play (optional): initialization domain.
- MM_DOMAIN_VMGM — Video Manage Menu Domain (optional): The following functionality operates in this domain:
  - title menu
  - legal notices and warnings
  - previews (occasionally)
- MM_DOMAIN_VTSM — Video Title Set Menu Domain (optional). Most menus operate in this domain:
  - root menu
  - PTT (chapter selection) menu

  - audio menu
  - sub-picture (subtitles) menu
  - angle menu

- MM_DOMAIN_TT — Title Domain (*mandatory*).This domain includes most previews, the main feature, etc., and is usually in standard (playback) mode.

Classification:

QNX Multimedia

See also:

mme_dvd_get_status(), mme_video_get_status.html()

Synopsis:

```
#include <mme/types.h>

typedef struct mm_media_status {
    uint32_t    title;
    uint32_t    title_count;
    uint32_t    chapter;
    uint32_t    chapter_count;
    uint32_t    num_audio_streams;
    uint32_t    audio_stream;
    uint32_t    num_subtitle_stream
    uint32_t    subtitle_stream;
    uint32_t    num_angles;
    uint32_t    angle;
} mm_media_status_t;
```

Description:

The structure **mm_media_status** carries information about a media device, such as an iPod, that also serves as a mediastore. It includes at least the members described in the table below.

Member	Type	Description
title	**uint32_t**	The currently playing media title.
title_count	**uint32_t**	The number of the current title.
chapter	**uin32_t**	The media title's currently playing chapter.
chapter_count	**uint32_t**	The number of the current chapter.
num_audio_streams	**uin32_t**	The number of available audio streams.
audio_stream	**uin32_t**	The current audio stream.
num_subtitle_streams	**uin32_t**	The number of available subtitle streams.
subtitle_stream	**uin32_t**	The current subtitle stream.
num_angles	**uin32_t**	The number of available angles.
angle	**uin32_t**	The current angle.

mm_media_status_event_t

```
typedef struct mm_media_status_event {
    mm_media_status_t       status;
    mm_media_status_reason_t  reason;
} mm_media_status_event_t;
```

The structure **mm_media_status_event_t** carries media information delivered with a MME_EVENT_MEDIA_ event, including its status, in **mm_media_status_t**, and the reason for the status event delivery, in **mm_media_status_reason_t**. It includes at least the members described in the table below.

Member	Type	Description
status	**struct**	Information about a media.
reason	**enum**	The reason for the media event delivery.

mm_media_status_reason_t

```
typedef enum mm_media_status_reason {
    MM_MEDIA_TITLE_UPDATE
    MM_MEDIA_CHAPTER_UPDATE
    MM_MEDIA_ANGLE_UPDATE
    MM_MEDIA_AUDIO_UPDATE
    MM_MEDIA_SUBTITLE_UPDATE
} mm_media_status_reason_t;
```

The enumerated type **mm_media_status_reason_t** is used to indicate the reason for which a media status update is delivered. It can be set to the following values:

- MM_MEDIA_TITLE_UPDATE (0x01)— the media title has changed.
- MM_MEDIA_CHAPTER_UPDATE (0x02) — th media chapter has changed.
- MM_MEDIA_ANGLE_UPDATE (0x04) — the media angle has changed.
- MM_MEDIA_AUDIO_UPDATE (0x08) — the media audio stream has changed.
- MM_MEDIA_SUBTITLE_UPDATE (0x10) — the media subtitle stream has changed.

Classification:

QNX Multimedia

See also:

mme_device_get_conf(), mme_device_set_conf()

Synopsis:

```
#include <mm/types.h>

typedef struct mm_metadata {
    const char *strings[ MM_METADATA_NUM_STRINGS ];
    const char *reserved1[ MM_METADATA_TOTAL_STRINGS - MM_METADATA_NUM_STRINGS ];
    uint16_t   release_year;
    uint8_t    release_month;
    uint8_t    release_mday;
    uint16_t   track_num;
    uint16_t   disc_num;
    uint32_t   reserved2[ 4 ];
} mm_metadata_t;
```

Description:

The structure **mm_metadata_t** carries video metadata. Its members include at least those listed in the table below.

Member	Type	Description
**strings*	**const char**	Array of pointers to video metadata; the number of pointers is set by the constant MM_METADATA_NUM_STRINGS. See **mm_metadata_string_index_t** below.
**reserved1*	**const char**	Reserved array size; the number of pointers is equal to MM_METADATA_TOTAL_STRINGS minus MM_METADATA_NUM_STRINGS. Reserved for future use.
release_year	**uint16_**	The year the media content was released
release_month	**uint8_t**	The month the media content was released
release_mday	**uint8_t**	The day of the month the media content was released.
track_num	**uint16_t**	The track number on the mediastore.
disc_num	**uint16_t**	The disk number of the media store.
reserved2	**uint32_t**	Reserved for future use.

mm_metadata_string_index_t

The enumerated type **mm_metadata_string_index_t** is used to index the strings inside the structure **mm_metadata_t**. Its values include:

- MM_METADATA_TITLE
- MM_METADATA_ARTIST

- MM_METADATA_COMPOSER
- MM_METADATA_ALBUM
- MM_METADATA_GENRE
- MM_METADATA_COMMENT
- MM_METADATA_NUM_STRINGS — the total number of pointers available to the member *string* in the structure **mm_metadata_t**.
- MM_METADATA_TOTAL_STRINGS — (16)

The enumerated type **mm_metadata_string_index_t** is used when accessing the *strings* member of an **mm_metadata_t** type. For example:

```
mm_metadata_t metadata;
char *artist;
mme_version_of_get_metadata_function(&metadata);
// print out the artist...
printf ("Artist is %s\n", artist = metadata.strings[MM_METADATA_ARTIST]? artist, "Unknown");
```

Classification:

QNX Multimedia

See also:

mm_video_info_t, **mm_video_audio_info_t**

Synopsis:

```
#include <mm/types.h>

enum mm_subpict_lang_ext;
```

Description:

The enumerated type **mm_subpict_lang_ext** defines the video language extension codes for audio streams and subtitles. Its values include:

- MM_NOT_SPECIFIED
- MM_CAPTION_NORMAL
- MM_CAPTION_LARGE
- MM_CAPTION_CHILDRENS
- MM_CLOSED_CAPTION_NORMAL
- MM_CLOSED_CAPTION_LARGE
- MM_CLOSED_CAPTION_CHILDRENS
- MM_CAPTION_FORCED
- MM_DIRETORS_COMNENT_NORMAL
- MM_DIRECTORS_COMMENT_LARGE
- MM_DIRECTORS_COMMENT_CHILDREN

Classification:

QNX Multimedia

See also:

mm_video_audio_info_t, **mm_video_subtitle_info_t**

Synopsis:

```
#include <mm/types.h>

enum mm_uop_t;
```

Description:

The enumerated type **mm_uop_t** defines User Operations Prohibitions values. These values are listed below:

- MM_UOP_CLOSE — prohibit application close.
- MM_UOP_GET_BOOKMARK — prohibit access to bookmarks.
- MM_UOP_SET_BOOKMARK — prohibit
- MM_UOP_GET_SPRMS — prohibit access to system parameter registers.
- MM_UOP_GET_GPRMS — prohibit access to general parameter registers.
- MM_UOP_SET_GPRM — prohibit modification of general parameter registers.
- MM_UOP_STOP — prohibit stop playback.
- MM_UOP_GO_UP — prohibit go up.
- MM_UOP_PREV_PG_SEARCH — prohibit searching to previous entity in program chain (typically search to previous chapter).
- MM_UOP_TOP_PG_SEARCH — prohibit searching to first entity in program chain (typically search to first chapter).
- MM_UOP_NEXT_PG_SEARCH — prohibit searching to next entity in program chain (typically search to next chapter).
- MM_UOP_SET_SPEED — prohibit set speed.
- MM_UOP_FRAME_ADVANCE — prohibit frame advance.
- MM_UOP_FRAME_REVERSE — prohibit frame reverse.
- MM_UOP_RESUME — prohibit resume playback.
- MM_UOP_UPPER_BUTTON_SELECT — prohibit upper button selection.
- MM_UOP_LOWER_BUTTON_SELECT — prohibit lower button selection.
- MM_UOP_LEFT_BUTTON_SELECT — prohibit left button selection.
- MM_UOP_RIGHT_BUTTON_SELECT — prohibit right button selection.

- MM_UOP_BUTTON_ACTIVATE — prohibit button activation.
- MM_UOP_BUTTON_SELECT_AND_ACTIVATE — prohibit button selection and activation.
- MM_UOP_STILL_OFF — prohibit turing still mode off.
- MM_UOP_PAUSE_ON — prohibit turning pause on.
- MM_UOP_PAUSE_OFF — prohibit turning pause off.
- MM_UOP_MENU_LANGUAGE_SELECT — prohibit selection of language menu.
- MM_UOP_AUDIO_STREAM_CHANGE — prohibit changing the audio stream.
- MM_UOP_SUB_PICTURE_STREAM_CHANGE — prohibit changing the subtitle stream.
- MM_UOP_ANGLE_CHANGE — prohibit changing the angle.
- MM_UOP_VIDEO_MODE_CHANGE — prohibit changing the video mode.
- MM_UOP_BUTTON_SELECT — prohibit button selection.
- MM_UOP_BUTTON_SELECT_POINT — prohibit selection of button by coordinates.
- MM_UOP_BUTTON_ACTIVATE_POINT — prohibit activation of button my coordinates (i.e. by pressing *chapter* on track screen).
- MM_UOP_SUB_PICTURE_STREAM_CHANGE_STREAM — prohibit changing the subtitle stream.
- MM_UOP_SUB_PICTURE_STREAM_CHANGE_DISPLAY — prohibit turning subtitles on or off.
- MM_UOP_AUDIO_LANGUAGE_SELECT — prohibit selection of the audio language.
- MM_UOP_SUB_PICTURE_LANGUAGE_SELECT — prohibit changing the subtitle language.
- MM_UOP_REPEAT_MODE_CHANGE — prohibit changing the repeat more.
- MM_UOP_TITLE_PLAY — prohibit playing the entire title.
- MM_UOP_PTT_PLAY — prohibit part of title play (i.e. jump to a title or chapter).
- MM_UOP_TITLE_TIME_PLAY — prohibit play from at time in the title (i.e. jump to a time in the title).
- MM_UOP_TITLE_TIME_SEARCH — prohibit search to a specific time in the title.
- MM_UOP_PTT_SEARCH — prohibit search by part of chapter.

- MM_UOP_MENU_CALL_VIDEO — prohibit jump to video menu.
- MM_UOP_PARENTAL_LEVEL_SELECT — prohibit selection of parental control level.
- MM_UOP_PARENTAL_COUNTRY_SELECT — prohibit country selection for parental control.
- MM_UOP_KARAOKE_MODE_CHANGE — prohibit changing karaoke mode.
- MM_UOP_PTT_PLAY_RANGE — prohibit playback of part of title, by range of chapters.
- MM_UOP_TITLE_TIME_PLAY_RANGE — prohibit playback of part of title, by time range.
- MM_UOP_FIRST_PLAY — prohibit playback of first title.
- MM_UOP_TITLE_GROUP_PLAY — prohibit playback by group.
- MM_UOP_TRACK_PLAY — prohibit playback by track.
- MM_UOP_GROUP_TIME_PLAY — prohibit playback of group of titles by time.
- MM_UOP_GROUP_TIME_SEARCH — prohibit searching for time in a group.
- MM_UOP_TRACK_SEARCH — prohibit searching for specific tracks.
- MM_UOP_PREV_TK_SEARCH — prohibit searching for previous track.
- MM_UOP_TOP_TK_SEARCH — prohibit searching for top track.
- MM_UOP_NEXT_TK_SEARCH — prohibit searching for next track.
- MM_UOP_PREV_DLIST_SEARCH — DVD-audio only: prohibit jumping to previous playlist.
- MM_UOP_NEXT_DLIST_SEARCH — DVD-audio only: prohibit jumping to next playlist.
- MM_UOP_HOME_DLIST_SEARCH — DVD-audio only: prohibit jumping to first playlist.
- MM_UOP_MENU_CALL_AUDIO — DVD-audio only: prohibit jumping to DVD-audio menu.
- MM_UOP_TEXT_LANGUAGE_SELECT — DVD-audio only: prohibit text language selection.
- MM_UOP_HIDDEN_GROUP_PLAY — DVD-audio only: prohibit playback of of hidden groups.
- MM_UOP_HIDDEN_TRACK_PLAY — DVD-audio only: prohibit playback of hidden tracks.
- MM_UOP_HIDDEN_TIME_PLAY — DVD-audio only: prohibit playback of hidden time.

Classification:

QNX Multimedia

See also:

`mm_video_info_t`

mm_video_angle_info_t

Video angle settings

Synopsis:

```
#include <mm/types.h>

typedef struct mm_video_angle_info {
    uint32_t      title;
    uint8_t       total;
    int8_t        current;
    int8_t        angles_available;
    int8_t        align;
} mm_video_angle_info_t;
```

Description:

The structure **mm_video_angle_info_t** includes at least the members described in the table below.

Member	Type	Description
title	**uint32_t**	The title of video for which angle information is provided.
total	**uint8_t**	The number of video angles available.
current	**int8_t**	The current video angle.
angles_available	**int8_t**	Indicate if changing the video angle will take effect on the current chapter. Clear if no effect on the current chapter.
align	**int8_t**	Aligns the structure to 32 bits.

Classification:

QNX Multimedia

See also:

mme_video_get_status(), mme_video_set_angle()

Synopsis:

```
#include <mm/types.h>

typedef struct mm_video_audio_info {
    uint32_t    title;
    int8_t      total;
    int8_t      current;
    struct mm_audio_attr {
        char       lang[2];
        uint8_t    ext;
        uint8_t    type;
        uint8_t    channels;
        uint8_t    spare;
    } attr[MM_MAX_VIDEO_AUDIO_STREAMS];
} mm_video_audio_info_t;
```

Description:

The structure **mm_video_audio_info_t** structure carries information about the languages of a video's subtitles. It includes at least the members described in the table below.

Member	Type	Description
title	**uint32_t**	The title for which audio stream information is provided.
total	**int8_t**	The number of available audio streams. If this field is 0 (zero), no audio streams are available.
current	**int8_t**	The audio stream currently selected. If this field is set to -1, no audio is currently playing.
attr	**struct**	An array of structures: **mm_audio_attr_t**, of length MM_MAX_AUDIO_STREAMS, containing audio languages information.

mm_audio_attr_t

The structure **mm_audio_attr_t** carries information about the languages of a video's audio streams. It includes at least the members described in the table below.

Member	Type	Description
lang	**char**	Two-character ISO 639-1 language code for the audio stream.
ext	**uint8_t**	Language extension codes. See **mm_subpict_lang_ext** in this reference.
type	**uint8_t**	Audio stream type.
channels	**uint8_t**	Total number of audio channels, including a low frequency channel. For example, 8 = 7.1, 6 = 5.1, 3 = 2.1, 4 = 4, 2 = 2, 1 = 1, and 255 = “unknown”.
spare	**uint8_t**	Unused

Classification:

QNX Multimedia

See also:

mme_video_get_audio_info(), mme_video_set_audio()

Synopsis:

```
#include <mm/types.h>
typedef struct mm_video_info {
    struct {
        uint16_t w;
        uint16_t h;
    } aspect_ratio;
    uint32_t    width;
    uint32_t    height;
    uint32_t    capture_format;
    uint32_t    frame_width;
    uint32_t    frame_height;
    uint32_t    max_bufferable_frames;
    uint32_t    display_mode;
    uint32_t    flags;
    char        codec[32];
} mm_video_info_t;
```

Description:

The structure **mm_video_info_t** provides information about a video. It includes at least the members described in the table below.

Member	Type	Description
aspect_ratio	**struct**	The width to height aspect ratio of the video. See **aspect_ratio** below.
width	**uint32_t**	The width of the video source, in pixels.
height	**uint32_t**	Height of the video source, in pixels.
capture_format	**uint32_t**	Flags for capturing additional information useful for presenting the video. See *video_flags* below.
frame_width	uint32_t	The width, in pixels, of the rendered video in video memory; may be smaller than the frame width. A value different from *width* does not imply scaling; see "*flags*" below.
frame_height	uint32_t	The height, in pixels, of the rendered video in video memory; may be smaller than the frame height. A value different from *width* does not imply scaling; see "*flags*" below.

continued...

Member	Type	Description
max_bufferable_frames	uint32_t	The maximum number of frames that can be requested for buffering by a call to the function *mme_video_set_properties()*. A -1 indicates that the video player does not support bufferable frames.
display_mode	**uint32_t**	The video display mode. See **mm_display_mode**
flags	uint32_t	Flags indicating how to handle the video display frame croppiing and scaling.
codec	**char**	A character string with name of the video codec. See "Video codec" below.

aspect_ratio

The **aspect_ratio** member uses whole numbers to express the video aspect ratio. These numbers only describe the height to width *ratio* of the image, and have no bearing on the actual width and height in pixels of the source.

Common aspect ratio values are:

- 235:100 (2.35:1)
- 16:9 or 166:100 (1.66:1) and (4/3)

Usual representations are in parentheses: "(*x*,*y*)".

w and *h*

The *w* and *h* members of the structure **aspect_ratio** are the whole numbers used to express the aspect ratio of the image.

Width *w* and height *h* values of 0 (0,0) mean that no aspect ratio information is available.

width and *height*

The *width* and *height* are the actual width and height of the source image, *in pixels*.

flags

The *flags* member of the structure **mm_video_info_t** uses the following values:

- MM_VIDEO_SOURCE_CROP — the video player can crop the source video and render only the cropped content to the video memory.
- MM_VIDEO_SCALEABLE — the video player can scale (or zoom) the specified source video and place the scaled result in video memory; if this flag is not set, the *video_width* and *video_height* members describe the active video dimensions.

- MM_VIDEO_FRAME_SETABLE — the video player can adjust the video memory image size.
- MM_VIDEO_SOURCE_PICTURE_LETTERBOXED — a 4:3 source picture; if the source picture is 16:9, black bars are added to make the picture 4:3.
- MM_VIDEO_AUTO_SCALED — the video is scaled to best fit the frame described in **mm_video_info_t**.

Video codec

The function *video_get_status()* uses the data structure **mm_video_info_t**. The function *mme_audio_get_status()* uses the data structure **mm_audio_format_t**. Both these structures include a member *codec*.

The codec members of the structures **mm_video_info_t** and **mm_audio_format_t** hold character strings identifying the codec format for the video or audio. These strings can have a length of up to the number of bytes defined by MM_CODEC_NAME_MAX_LEN, which is currently 32 bytes.

Client applications can pass these character strings up to the end users to inform them of the codec format used by a video or audio track.

mm_display_mode

The enumerated type **mm_display_mode** describes a video's display mode. Its values include:

- MM_DISPLAY_MODE_NORMAL
- MM_DISPLAY_MODE_LETTERBOX
- MM_DISPLAY_MODE_PANSCAN
- MM_DISPLAY_MODE_OPEN_MATTE

capture_format

The enumerated type **capture_format** describes a video's capture format. Its values include:

- MM_CAPTURE_NTSC
- MM_CAPTURE_PAL
- MM_CAPTURE_OTHER

Classification:

QNX Multimedia

See also:

mm_audio_format_t, **mm_bitrate_t**, **mm_video_audio_info_t**, **mme_video_properties_t**, **mme_audio_get_status**, *mme_video_get_status()*

Synopsis:

```
#include <mm/types.h>

typedef struct mm_video_properties {
    uint32_t   flags;
    struct {
        uint32_t  left,top,right,bottom;
    } source;
    struct {
        uint32_t  left,top,right,bottom;
    } dest;
    uint32_t   frame_width;
    uint32_t   frame_height;
    uint32_t   frame_buffers;
    uint32_t   display_mode;
} mm_video_properties_t;
```

Description:

The structure **mm_video_properties_t** describes video display properties. It includes at least the members described in the table below.

Member	Type	Description
flags	**uint32_t**	Flags indicating how to handle the video display.
source	**struct**	The rectangle (left and top inclusive; right and bottom exclusive) to extract from the source video; must be within the *width* and *height* dimensions given by *mme_video_get_info()*; it is ignored if MM_AUTO_SCALE is set.
dest	**struct**	The rectangle (left and top inclusive; right and bottom exclusive) to render the video into; it must be within the *frame_width* and *frame_height* dimensions given by *mme_video_get_info()*; it is ignored if MM_AUTO_SCALE is set.
frame_width	**uint32_t**	Specify the width, in pixels, of the video surface to use when rendering a video; it does not imply scaling (the frame may or may not be completely filled by the rendered video); it is used only if the MM_SET_VID_FRAME_SIZE flag is set.

continued...

Member	Type	Description
frame_height	**uint32_t**	Specify the height, in pixels, of the video surface to use when rendering a video; it does not imply scaling (the frame may or may not be completely filled by the rendered video); it is used only if the MM_SET_VID_FRAME_SIZE flag is set.
frame_buffers	**uint32_t**	Specify the number of video frames to buffer; must be less than or equal to *max_bufferable_frames* given by *mme_video_get_info()*; it is only used if the MM_SET_FRAME_BUFFERS flag is set.
display_mode	**uint32_t**	The video display mode; used only if the MM_SET_DISPLAY_MODE flag is set.

For more information about video dimensions and aspect ratio see **mm_video_info_t**.

Currently **io-media-generic** only supports setting the video source and destination (the *source* and *dest* members of the **mm_video_properties_t** structure). Other **io-media** variants may support other capabilities.

left, *top*, *right* and *bottom*

The *left*, *top*, *right* and *bottom* members of the structures **source** and **dest** define, respectively, the video source and destination video rectangles, in pixels. The *left* and *top* values are inclusive; the *right* and *bottom* values are exclusive.

flags

The *flags* member of the structure **mm_video_properties_t** uses the following values:

- MM_AUTO_SCALE — ask the player to determine how best to display the video; if this flag is set, *source* and *dest* members are ignored.
- MM_SET_VID_FRAME_SIZE — set to use the values in the *frame_width* and *frame_height* members. If this flag is *not* set, the *frame_width* and *frame_height* members are ignored.
- MM_SET_FRAME_BUFFERS — use the values in the *frame_buffers* member. If this flag is *not* set the *frame_buffers* member is ignored.
- MM_SET_DISPLAY_MODE — use the values in the *display_mode* member. Use this flag only if MM_AUTO_SCALE is set. If this flag is *not* set, the *display_mode* member is ignored.

Classification:

QNX Multimedia

See also:

mm_audio_format_t, **mm_audio_type**, **mm_bitrate_t**, **mme_video_audio_info_t**, **mme_video_info_t**, **mme_audio_get_status**, *mme_video_get_status()*, *mme_video_get_info()*, *mme_video_set_properties()*

mm_video_status_t

Video status information

Synopsis:

```
#include <mm/types.h>

typedef struct mm_video_status {
    uint32_t    width;
    uint32_t    height;
    struct {
        uint16_t w;
        uint16_t h;
    } aspect_ratio;
} mm_video_status_t;
```

Description:

The structure **mm_video_status_t** describes a video's status. It is filled in by the function *mme_video_get_status()* and includes at least the members described in the table below.

Member	Type	Description
width	**uint32_t**	The width of the video, in pixels.
height	**uint32_t**	Height of the video, in pixels.
aspect_ratio	**struct**	The width to height aspect ratio of the video. See **aspect_ratio** below.

aspect_ratio

The **aspect_ratio** member uses whole numbers to express the video aspect ratio. These numbers only describe the height to width *ratio* of the image, and have no bearing on the actual width and height in pixels of the source.

Common aspect ratio values are:

- 235:100 (2.35:1)
- 16:9 or 166:100 (1.66:1) and (4/3)

Usual representations are in parentheses: "(*x*,*y*)".

w and *h*

The *w* and *h* members of the structure **aspect_ratio** are the whole numbers used to express the aspect ratio of the image.

Width *w* and height *h* values of 0 (0,0) mean that no aspect ratio information is available.

Classification:

QNX Multimedia

See also:

mme_video_get_info(), mme_video_get_status()

mm_video_subtitle_info_t

Video subtitle and caption information

Synopsis:

```
#include <mm/types.h>

typedef struct mm_video_subtitle_info {
    uint32_t    title;
    uint8_t     total;
    int8_t      current;
    struct mm_video_subtitle_attr {
        char       lang[2];
        uint8_t    ext;
    } attr[MM_MAX_VIDEO_SUBTITLES];
} mm_video_subtitle_info_t;
```

Description:

The structure **mm_video_subtitle_info_t** carries information about a video's subtitles. It includes at least the members described in the table below.

Member	Type	Description
title	**uint32_t**	The title of video for which subtitle information is provided.
total	**uint8_t**	The number of available subtitles. If this field is 0 (zero), no subtitles are available.
current	**int8_t**	The current subtitle, which is in the range of 0 to *total* - 1 (number of available subtitles). If this field is set to -1, no subtitles are currently displayed.
attr	**array**	An array of structures: **mm_video_subtitle_attr_t**, of length MM_MAX_VIDEO_SUBTITLES, containing subtitle languages information.

mm_video_subtitle_attr_t

The structure **mm_video_subtitle_attr_t** contains information about the languages of a video's subtitles. It includes at least the members described in the table below.

Member	Type	Description
lang	**array**	An array with two-character ISO 639-1 language codes for the subtitle.
ext	**uint8_t**	Language extension codes. See **mm_subpict_lang_ext**.

Classification:

QNX Multimedia

See also:

mm_subpict_lang_ext, *mme_video_get_subtitle_info()*, *mme_video_set_subtitle()*

mme_audio_get_status()

Get the audio status

Synopsis:

```
#include <mme/mme.h>

int mme_audio_get_status ( mme_hdl_t *hdl,
                           mm_audio_format_t *status );
```

Arguments:

hdl An MME connection handle.

status A pointer to a **mm_audio_format_t** structure that the function fills in with information about the audio stream for the current track.

Library:

mme

Description:

The function *mme_audio_get_status()* gets audio stream information for the currently playing track and places it in *status*. See **mm_audio_format_t** in this reference.

Events

None delivered.

Blocking and validation

This function blocks on the control context and on **io-media**. It does not validate any data, and returns with either the requested information or an error.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mm_audio_format_t, *mme_video_get_status()*

mme_bookmark_create()

Create a bookmark for the playing track

Synopsis:

```
#include <mme/mme.h>

int mme_bookmark_create( mme_hdl_t *hdl,
                         const char *name,
                         uint64_t *bookmarkid );
```

Arguments:

hdl	An MME connection handle.
name	The bookmark name. Set to NULL if the bookmark name is not important.
bookmarkid	The bookmark ID.

Library:

mme

Description:

The function *mme_bookmark_create()* creates a bookmark on a playing track at the current point of the playback.

Bookmarks allow end users to mark points in tracks from which they want to resume playing these tracks. They are used by *mme_play_bookmark()*, which starts playback of a track in a track session at the specified bookmark instead of at their beginning.

Events

None delivered.

Blocking and validation

This function behaves as follows, depending on the MME connection:

- Synchronous — fully validating and blocks on **io-media**.
- Asynchronous — replies before the bookmark is created; it doesn't block on **io-media**.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_bookmark_delete(), mme_play_bookmark()

mme_bookmark_delete()

Delete a bookmark from a track

Synopsis:

```
#include <mme/mme.h>

int mme_bookmark_delete( mme_hdl_t *hdl,
                         uint64_t bookmarkid,
                         uint64_t fid );
```

Arguments:

hdl	The MME connection handle.
bookmarkid	The bookmark ID. Set this argument to **0** if you are deleting the bookmark(s) based on the file ID (*fid*).
fid	The ID for the file from which you want to delete all bookmarks. Set this argument to **0** and use *bookmarkid* if you want to delete only one, specified bookmark from the file.

Library:

mme

Description:

The function *mme_bookmark_delete()* deletes a specified bookmark or all bookmarks on a specified track. Note that you can specify either *bookmarkid* to delete a specific bookmark, or *fid* to delete all bookmarks for a specified track, but you can *not* specify both *bookmarkid* and *fid*.

Events

None delivered.

Blocking and validation

This function is fully validating and runs to completion.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_bookmark_create(), mme_play_bookmark()

mme_buffer_status_t

Buffer status information

Synopsis:

```
#include <mme/types.h>

typedef struct mme_buffer_status {
    uint32_t    state;
    uint32_t    read_ms;
    uint32_t    max_ms;
    uint32_t    reserved;
} mme_buffer_status_t;
```

Description:

The data structure **mme_buffer_status_t** carries buffer status information. Its members are described in the table below.

Member	Type	Description
state	**uint32_t**	The buffer state. See **mme_buffer_state_t** below.
read_ms	**uint32_t**	The number of milliseconds of playback time that are currently in the buffer.
max_ms	**uint32_t**	The maximum buffer size, in milliseconds.
reserved	**uint32_t**	Reserved for internal use.

The value in *read_ms* can be higher than the value in *max_ms*. Values are rounded *up* to the nearest MRA buffer size, so the current buffer level can be reported as greater that the set level.

mme_buffer_state_t

The enumerated type **mme_buffer_state_t** defines buffer states as follows:

- MME_BUFFER_STATE_NORMAL (0) — the MME is playing from the buffer and draining it, but is not reading anything into the buffer.
- MME_BUFFER_STATE_PREFETCHING (1) — the MME is reading a track and filling the buffer, but there is not enough playback time in the buffer to start playback.
- MME_BUFFER_STATE_BUFFERING (2) — the MME is both reading a track and filling the buffer, and playing from the buffer and draining it.

Classification:

QNX Multimedia

See also:

mme_time_t, **mme_playstate_t**, **mme_playstate_speed_t**

Synopsis:

```
#include <mme/mme.h>

int mme_button( mme_hdl_t *hdl,
                mm_button_t button );
```

Arguments:

hdl An MME connection handle.

button The "button" command to pass to the MME in the enumerated type **mm_button_t**.

Library:

mme

Description:

The function *mme_button()* passes button commands for navigable tracks from your client application to the MME. A *navigable* track is one of the following:

- a track, such as DVD video, that contains a built-in menu
- a track on a device, such as an iPod, that has its own navigation interface

Using the *mme_button()* function with an iPod device

iPods manage their own track sessions. To move to the next or previous track in an iPod track session, call the *mme_button()* function with **mm_button_t** set to MM_BUTTON_NEXT or MM_BUTTON_PREV, as required.

Checking if a device can manage its own track sessions

To check is a device can manage its own track sessions, the client application can call *mme_play_get_info()* to get the data structure **mme_play_info_t**. If the *support* flag contains MME_PLAYSUPPORT_DEVICE_TRACKSESSION, the current device manages its own track sessions.

Using Repeat and Repeat AB modes

The MM_BUTTON_REPEAT_OFF and MM_BUTTON_REPEAT_AB_OFF values can be used together to add repeat functionality. For example, you can repeat the current title, then while repeating the title, mark A and mark B and repeat the AB range. You can then turn off the repeat AB mode, leaving the repeat title mode active; or, you can turn off the repeat title mode, leaving the repeat AB mode active.

mm_button_t

The enumerated type **mm_button_t** defines the button command to pass to the MME. It can be set to any of the values listed in the table below.

Note that button commands only work for devices with navigable tracks, as described above (iPod, DVD-V, Bluetooth), and that most devices only support a subset of the functionality listed in the table. Check the table to see which devices support which button values.

Value	iPod	Bluetooth	DVD-V	Action
MM_BUTTON_NEXT	Y	Y	Y	Skip to next track.
MM_BUTTON_PREV	Y	Y	Y	Skip to previous track.
MM_BUTTON_TOP	N	N	Y	Skip to first track.
MM_BUTTON_CURSOR_LEFT	N	N	Y	Move cursor left.
MM_BUTTON_CURSOR_RIGHT	N	N	Y	Move cursor right.
MM_BUTTON_CURSOR_UP	N	N	Y	Move cursor up.
MM_BUTTON_CURSOR_DOWN	N	N	Y	Move cursor down.
MM_BUTTON_ENTER	N	N	Y	Activate the currently highlighted item.
MM_BUTTON_RETURN	N	N	Y	Return to previous activity (i.e. playback). This button is equivalent to MM_BUTTON_RESUME
MM_BUTTON_GOUP	N	N	Y	See MM_BUTTON_GOUP below.
MM_BUTTON_MENU_TITLE	N	N	Y	Show title menu.
MM_BUTTON_MENU_ROOT	N	N	Y	Go to root menu.
MM_BUTTON_MENU_AUDIO	N	N	Y	Show audio properties menu.
MM_BUTTON_MENU_ANGLE	N	N	Y	Show video angle menu.
MM_BUTTON_MENU_SUBTITLE	N	N	Y	Show subtitle menu.
MM_BUTTON_MENU_PTT	N	N	Y	Show title or chapter menu.
MM_BUTTON_REPEAT_AB_OFF	N	N	Y	Turn repeat from point A to B off. See Using Repeat and Repeat AB modes below.
MM_BUTTON_REPEAT_AB_POINT_A	N	N	Y	Set repeat point A.
MM_BUTTON_REPEAT_AB_POINT_B	N	N	Y	Set repeat point B.

continued...

Value	iPod	Bluetooth	DVD-V	Action
MM_BUTTON_REPEAT_OFF	N	N	Y	Turn repeat mode off.
MM_BUTTON_REPEAT_CHAPTER	N	N	Y	Repeat current chapter.
MM_BUTTON_REPEAT_TITLE	N	N	Y	Repeat current title.
MM_BUTTON_REPEAT_DISC	N	N	Y	Repeat current disc.
MM_BUTTON_RESUME	N	N	Y	Resume previous activity (i.e. playback).
MM_BUTTON_FRAME_ADVANCE	N	N	Y	Advance to next video frame.
MM_BUTTON_FRAME_REVERSE	N	N	Y	Move to previous video frame.
MM_BUTTON_PAUSE	N	Y	Y	Pause play.
MM_BUTTON_PLAY	N	Y	Y	Play.
MM_BUTTON_STOP	N	Y	Y	Stop play.
MM_BUTTON_0 to 99	N	N	Y	Accept input from button *n* on a remote control.

DVD, and video support is platform specific, and the current MME release supports DVD mediastores and video playback only with custom **`io-media`** modules. Similarly, Bluetooth support is scheduled for a future release, or custom implementations.

If MME API functions that support DVD mediastores and video playback are called on a system that does not have the required **`io-media`** modules, these functions return **`-1`** and set *errno* to ENOSYS.

Please contact QNX to discuss your implementation requirements.

MM_BUTTON_GOUP

The behavior of MM_BUTTON_GOUP is determined by the author of the DVD. Typically, this button is used to jump to the start of the context the user is in. For example, if the user is playing a movie, this button jumps to the start of the movie; or, if the user is in a fourth level menu, this button jumps to the topmost menu.

Events

This function may return playback error events: MME_PLAY_ERROR_* andMME_EVENT_PLAY_ERROR.

Blocking and validation

This function verifies that the client application code is valid. It blocks on control contexts.

If *mme_button()* is called and another function is called before *mme_button()* returns, the second function blocks on **`io-media`** until *mme_button()* returns. If there are no other pending calls, *mme_button()* returns without blocking on **`io-media`**.

Returns:

≥**`0`**	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_get_info(), **`mme_play_info_t`**

mme_charconvert_setup()

Indicate the default character encoding

Synopsis:

```
#include <mme/mme.h>
#include <mme/charconver.h>

int mme_charconvert_setup( mme_hdl_t *hdl,
                           const char *default_encoding,
                           uint32_t allow_detection );
```

Arguments:

hdl	An MME connection handle.
default_encoding	A pointer to string passed to the character conversion DLL loaded into the MME. The contents of this string are not currently defined. The character conversion DLL must understand the contents of thsi string.
allow_detection	A flag that determines if the MME and the character conversion DLL are permitted to perform encoding detection. Set to 1 to allow detection, or to 0 to disable detection.

Library:

mme

Description:

The function *mme_charconvert_setup()* changes the default fallback character encoding and passes the new values to the **charconvert** DLL so that it knows the new values requested by the system.

Character encoding conversion is required to convert different multimedia sources (ID3, WMA, etc.) into UTF-8 character format, so that strings are consistent throughout the system.

The MME already provides the ability to extend its character conversion algorithms by using the external DLL **charconvert**. However, the DLL can MME communicate the encoding used by a media source to this DLL only if the source itself indicates that encoding. In cases where the media source does not provide character encoding information, the character conversion DLL must attempt to detect the encoding and, if it is unable to do so, fall back to a default encoding.

mme_charconvert_setup() makes setting of the fallback encoding dynamic to allow easy configuration for different areas of the world. A device controller can tell the MME what new default encoding to use, and the MME can in turn pass this information on to the character conversion DLL, which would uses that default.

Events

None delivered.

Blocking and validation

This function performs no validations and doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

"Creating an external DLL to provide character encoding routines" in the chapter Configuring Internationalization of the *MME Configuration Guide*.
mme_media_get_def_lang(), mme_media_set_def_lang()

Synopsis:

```
#include <mme/mme.h>

mme_hdl_t *mme_connect( const char *filename,
                        uint32_t flags );
```

Arguments:

filename The full pathname to the multimedia engine device name, including the control context (for example, **/dev/mme/control_context1**).

flags Flags that can be used to modify the behavior or the MME connection. See "Flags" below.

Library:

mme

Description:

The function *mme_connect()* connects the client application to the MME in a specified control context. It returns an **mme_hdl_t** object, which is used by the other *mme_*()* API functions.

To communicate to multiple control contexts you must use *mme_connect()* to connect at least once for each control context.

By default, the MME has one control context, but you can add more to the MME database, then connect to them. For more detailed information about control contexts, see "Connecting to the MME" in the *MME Developer's Guide*. For more information about the **controlcontexts** table, see the appendix MME Database Schema Reference.

CAUTION: Connections are not thread safe, so the client application must ensure that a connection handle isn't used by more than one thread at a time.

Device path

A control context's path maps directly to a resource manager device path. The device path, such as, for example, **/dev/mme/frontseat**, correlates directly to the control context with the same name; for example: "frontseat". The device may be on the same machine that the MME is running on, or it can be located on another machine accessible to the MME.

Flags

The client application can use the *flags* variable to configure the behavior of the MME connection. Behavior is configured as follows:

O_SYNC is **not** set (default).
: The MME returns to the client as soon as possible, and completes work after unblocking the client. It verifies the validity of as much of the request as possible before unblocking with a success code.

O_SYNC is set.
: The MME completely executes requests before returning to the client.

O_NONBLOCK is **not** set (default).
: The MME will block clients in a queue until it can service their requests.

O_NONBLOCK is set.
: The MME will return an error with *errno* set to EAGAIN if executing a client request would result in the client being blocked.

The blocking option is not honored by all MME functions. Synchronizations, for example, ignore the blocking flag and are always non-blocking. The main use for the non-blocking option is to give client application developers more control over the behavior of the MME playback functions.

Functions that use the QDB many block on the QDB.

Events

None delivered.

Blocking and validation

This function fully validates all data; all arguments are checked before the call returns. The operation is complete when the call returns.

Returns:

An initialized **`mme_hdl_t`**, or NULL if an error occurred (*errno* is set).

Examples:

The example below shows how to connect your client application to the MME:

```
#include <mme/mme.h>
#include <qdb/qdb.h>

static char *mme_device_name = "/dev/mme/default";
static char *qdb_device_name = "/dev/qdb/mme";

...

// Establish a connection to the QDB
// (to obtain information about tracks and their information)
if( NULL == (qdb = qdb_connect( qdb_device_name, 0 )) ) {
    fprintf( stderr, "%s: ", qdb_device_name );
    perror( "qdb_connect()" );
    exit( EXIT_FAILURE );
}

// Establish a connection to the MME
// (to control what to play)
if( NULL == (mme = mme_connect( mme_device_name, 0 )) ) {
    fprintf( stderr, "%s: ", mme_device_name );
    perror( "mme_connect()" );
    exit( EXIT_FAILURE );
}
```

Note that in the sample code above the *flags* variable is set to 0. The MME will use its default settings, which are O_SYNC and O_NONBLOCK not set.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

Caveats:

MME connections can be shared between threads in a process. However, they are not thread safe, so the client application must take precautions to ensure that the same connection handle isn't used by two threads at the same time.

See also:

mme_disconnect()

Synopsis:

```
#include <mme/types.h>

typedef struct _mme_copy_info {
   uint64_t             srcfid;
   uint64_t             dstfid;
   uint64_t             cqid;
} mme_copy_info_t;
```

Description:

The structure **mme_copy_info_t** carries information about media copying and ripping operations. The MME uses this structure with events such as MME_EVENT_MEDIACOPIER_COPYFID, MME_EVENT_MEDIACOPIER_SKIPFID and MME_EVENT_MEDIACOPIER_STARTFID to deliver information about the state of a media copy or ripping operation.

Member	Type	Description
srcfid	**uint64_t**	The file ID of the source file being copied or ripped.
dstfid	**uint64_t**	The file ID of the destination file.
cqid	**uint64_t**	The copy queue ID entry currently being copied or ripped.

Classification:

QNX Multimedia

See also:

mme_play_get_status(), “Event data” and the chapter Media Copy and Ripping Events

Synopsis:

```
#include <mme/mme.h>

int mme_delete_mediastores( mme_hdl_t *hdl,
                            uint32_t flags );
```

Arguments:

hdl An MME connection handle.

flags A flag determining if the function should delete mediastores marked as permanent. Set to a value defined by MME_DB_DELETION_*.

Library:

mme

Description:

The function *mme_delete_mediastores()* prunes from the MME database entries for mediastores whose state is **unavailable**. It deletes entries only for mediastores whose type (MME_STORAGETYPE_*) matches the storage types set by **<MediastoreMatching>** configuration elements. See “About pruning ejected mediastores”.

The function *mme_delete_mediastores()* can be called at any time, but it is usually used after a system startup to delete mediastores entries for mediastores whose states are set to **unavailable** because they were removed while the system was shut down. See Sample script **mme_del_unav** in the *MME Configuration Guide*.

The default behavior of *mme_delete_mediastores()* is to *not* delete entries for mediastores whose library entries mark them as permanent. However, you can set the *flag* argument to override this restriction and have *mme_delete_mediastores()* delete all entries for unavailable mediastores of the types permitted by the **<MediastoreMatching>** configuration elements.

The **<WhenUnavailable>** configuration has no affect on *mme_delete_mediastores()*.

MME_DB_DELETION_*

The MME defines the following values in **interface.h** that determine the behavior of *mme_delete_mediastores()*:

- MME_DB_DELETION_IGNORE_PERMANENT — (0x0001) delete the mediastore from the MME database, even if it or its **library** table entries are marked as permanent.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥`0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

*()*mme_resync_mediastore

mme_device_get_config()

Get device configuration information

Synopsis:

```
#include <mme/mme.h>

int mme_device_get_config( mme_hdl_t  *hdl,
                           uint64_t msid,
                           const char *xpath,
                           unsigned flags,
                           unsigned buflen,
                           char *buffer );
```

Arguments:

hdl	An MME connection handle.
msid	The mediastore ID of the device from which option information is required.
xpath	A pointer to the xpath of the XML element attribute to retrieve. This xpath must be the string **"/"** (Get all option configuration information).
flags	Flags to detemine the behavior of the operation. For future use.
buflen	The length of the buffer (referred to by *buffer*) for the device configuration.
buffer	A pointer to the buffer where the option option values are placed. See "Getting and setting device configuration values" in the *MME Developer's Guide* chapter External Devices, CD Changers and Streamed Media.

Library:

mme

Description:

The function *mme_device_get_config()* retrieves device configuration information for a specified device accessed through MediaFS.

Ensuring an adequate buffer length

The function *mme_device_get_config()* returns a buffer length when it successfully completes execution. This buffer length indicates only that the function did not fail. It does *not* indicate that the configuration information was successfully written to the buffer referenced by the *buffer* argument:

- If the value returned by *mme_device_get_config()* is less than or equal to (≤) the buffer length (*buflen*), the buffer was long enough for the requested information. The function wrote the information to the buffer and you can go on to the next operation.

- If the value returned by *mme_device_get_config()* is greater than (**>**) the buffer length (*buflen*), the buffer was too small for the requested information. You need to increase the buffer length to at least the returned value and call *mme_device_get_config()* again.

At present, *mme_device_get_config()* only supports:

- the following devices accessed through MediaFS:
  - iPod devices
  - Bluetooth devices using a Temic stack
- retrieving all option configuration information; individual elements or attributes can *not* be specified

Events

None delivered.

Blocking and validation

This function perfoms no validations and runs to completion.

Returns:

>0 The function completed successfully, but did not necessarily retrieve the requested information. See “Ensuring an adequate buffer length” above.

-1 An error occured (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

`mm_media_status_t`, *mme_device_set_config()*

mme_device_set_config()

Set device options

Synopsis:

```
#include <mme/mme.h>

int mme_device_set_config( mme_hdl_t *hdl,
                           uint64_t msid,
                           const char *xpath,
                           const char *newvalue,
                           unsigned flags );
```

Arguments:

hdl	An MME connection handle.
msid	The mediastore ID of the device from which option information is required.
xpath	A pointer to the xpath of the XML element attribute to set. This xpath must specify an XML element attribute; for example: **"/path/to/node@value"**.
newvalue	A pointer to the new value for the specified option.
flags	Flags to detemine the behavior of the operation. For future use.

Library:

mme

Description:

The function *mme_device_set_config()* sets a device configuration attribute for a specified device accessed through MediaFS.

As of this release, *mme_device_set_config()* only supports:

- iPod devices accessed through MediaFS
- setting a single option configuration attribute; you must call the function for each attribute you want to change

For more information, see “Getting and setting device configuration values” in the *MME Developer’s Guide* chapter External Devices, CD Changers and Streamed Media.

Events

None delivered.

Blocking and validation

This function performs no validations. It does not block.

Returns:

0 Success.

-1 An error occured (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

`mm_media_status_t`, *mme_device_get_config()*

mme_directed_sync_cancel()

Cancels a specified directed synchronization

Synopsis:

```
#include <mme/mme.h>

int mme_directed_sync_cancel( mme_hdl_t *hdl,
                              int operation_id );
```

Arguments:

hdl	An MME connection handle.
operation_id	The operation ID of the directed synchronization to be cancelled.

Library:

mme

Description:

The function *mme_directed_sync_cancel()* cancels a specified directed synchronization. The synchronization to cancel can be either in progress or pending.

To cancel a directed synchronization, set the parameter *operation_id* to the synchronization ID returned by *mme_sync_directed()*.

For more information about directed synchronizations, see *mme_sync_directed()*.

Events

This function can return synchronization error events (MME_SYNC_ERROR_*) and MME_EVENT_SYNCABORTED.

Blocking and validation

This function validates *operation_id* before returning.

Returns:

≥**0**	Success: the directed synchronization was cancelled, or the mediastore was not being synchronized when the cancellation request was made.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_resync_mediastore(), mme_setpriorityfolder(), mme_sync_directed(), mme_sync_file(), mme_sync_get_msid_status(), mme_sync_get_status()

Synopsis:

```
#include <mme/mme.h>

int mme_disconnect( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_disconnect()* disconnects the client application from the current MME control context.

If you want to disconnect from a control context but leave the MME process running and available for new client application connections, simply call *mme_disconnect()* with the handle of the control context from which you want to disconnect. However, if you want to shut down the MME, you must:

1. Call *mme_shutdown()* to stop playback and synchronization operations and prepare the MME for shutdown.
2. Call *mme_disconnect()* to disconnect from the MME.

For more information about how to shut down the MME, see *mme_shutdown()* and "Shutting down the MME" in the chapter Starting Up and Connecting to the MME of the *MME Developer's Guide*.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect(), mme_shutdown()

mme_dvd_get_disc region()

Get the region permissions for a DVD-video

Synopsis:

```
#include <mme/mme.h>

int mme_dvd_get_disc_region ( mme_hdl_t *hdl,
                              uint64_t msid,
                              uint32_t *region );
```

Arguments:

hdl	An MME connection handle.
msid	The ID of the mediastore (DVD-video disk) from which information is needed.
region	A pointer to the location where the function can store the region reported by the DVD-video disk.

Library:

mme

Description:

The function *mme_dvd_get_disc_region()* gets the region code of specific DVD-video disks that are inserted into the DVD drive. The bits set by *mme_dvd_get_disc_region()* represent the regions in which the DVD-video may be played. If no bits are set, the DVD-video is regionless and can be played in any region.

The *region* argument takes a 32-bit region code, but the top 24 bits of the region aren't currently used. Region codes are represented in bits 0 to 7, with bit 0 representing region 1, up to bit 7 representing region 8.

How to use *mme_dvd_get_disc_region()*

Before playing a DVD-video, the MME automatically checks the region for a DVD-video disk against the DVD drive region, and enforces permissions. If the user attempts to play a DVD-video in a drive that does not have permissions for that DVD-video's region, the MME generates a MME_PLAY_ERROR_REGION event.

You should use the function *mme_dvd_get_disc_region()* to check the regions of a DVD-video disk when you first access it. You can perform a bitwise AND operation to compare these regions against the region codes for which a device is enabled in order to determine if the DVD-video can be played on that device. For example, if the device is enabled for regions 1 and 3, you can check that a DVD-video disk is from one of these regions or has no region set before allowing the user to continue.

By getting the DVD-video disk regions on first access and checking these against the DVD drive regions, you can inform the end-user immediately in the event that the DVD-video is not playable on the drive.

Example: check if a DVD-video disk can be played on a device

```
/*
  * You can play a disk if it has no region (its region code
  * is 0), or if one of the disk region bits matches the
  * device region bits.
  */
if (bitsfromdisc == 0 || (bitsfromdisc & deviceregion) != 0) {
/* Region is OK */
}
```

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_status()

mme_dvd_get_status()

Get the status for a DVD

Synopsis:

```
#include <mme/mme.h>

int mme_dvd_get_status ( mme_hdl_t *hdl,
                         mm_dvd_status_t *status );
```

Arguments:

hdl An MME connection handle.

status A pointer to a **mm_dvd_status_t** structure the function fills in with information about the DVD status. See **mm_dvd_status_t**.

Library:

mme

Description:

The function *mme_dvd_get_status()* gets the status for a DVD device. This information is specific to DVD devices; for generic video playback information, use *mme_video_get_status()*.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_status(), **mm_dvd_status_t**

Synopsis:

```
#include <mme/explore.h>

int mme_explore_end( mme_explore_hdl_t *x_hdl);
```

Arguments:

x_hdl The explorer handle returned by *mme_explore_start()*.

Library:

mme

Description:

The function *mme_explore_end()* ends the exploration of an item on a media store.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_hdl_t, *mme_explore_info_free()*, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_hdl_t

The explorer API handle

Synopsis:

```
#include <mme/explore.h>

struct mme_explore_hdl;
typedef struct mme_explore_hdl mme_explore_hdl_t;
```

Description:

The structure **mme_explore_hdl_t** is used for exploration session control. One handle is used for each item explored.

Classification:

QNX Multimedia

See also:

mme_explore_end(), *mme_explore_info_free()*, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_info_free()

Free an explorer data structure

Synopsis:

```
#include <mme/explore.h>

int mme_explore_info_free( mme_hdl_t *hdl,
                           const mme_explore_info_t *info );
```

Arguments:

hdl A handle to the MME returned by *mme_explore_start()*.

info Pointer to the **mme_explore_info_t** structure to free.

Library:

mme

Description:

The function *mme_explore_info_free()* releases an **mme_explore_info_t** structure that was returned by *mme_explore_playlist_find_file()*, not in the context of an explorer session.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **mme_explore_hdl_t**, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_info_get()

Get information about an item being explored

Synopsis:

```
#include <mme/explore.h>

const mme_explore_info_t *mme_explore_info_get( mme_explore_hdl_t *x_hdl,
                                                uint32_t flags);
```

Arguments:

x_hdl An explorer handle returned by *mme_explore_start()*.

flags Flags describing the type of item.

Library:

mme

Description:

The function *mme_explore_info_get()* retrieves information about an item in a folder or a playlist file, and returns this information in the data structure **mme_explore_info_t**. This information is:

- The path and filename of the item.
- A flag describing the item (file, folder, playlist, etc.). See MME_EXPLORE_* bit masks in **mme_explore_hdl_t**.
- Metadata, if metadata has been requested. The default is to *not* retrieve metadata.

The path information is identical in format to the path information returned by *mme_ms_metadata_get()*, and used by *mme_play_file()* (deprecated).

The item the information is for is determined by:

- the current offset position in the folder
- the number of times this function has been called

Each time this function is called, the offset position is incremented by 1 (one), until either *mme_explore_end()* or *mme_explore_position_set()* is called. If no offset position is set, *mme_explore_info_get()* starts retrieving information from the first item in the folder.

Items retrieved by *mme_explore_info_get()* are presented as they occur; that is, they are *not* sorted or reorganized in any way.

MME_EXPLORE_RESOLVE_PLAYLIST_ITEM

The constant MME_EXPLORE_RESOLVE_PLAYLIST_ITEM is an inbound flag telling the MME to resolve playlist file entries immediately. Using this flag results in much faster resolution of playlist contents to playable files, but the actual playlist entry value is not visible at to the client application.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

An initialized **`mme_explore_hdl_t`**, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **`mme_explore_hdl_t`**, *mme_explore_info_free()*, **`mme_explore_info_t`**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_info_t

Information about items found during mediastore exploration

Synopsis:

```
#include <mme/explore.h>

typedef struct s_mme_explore_info {
uint32_t flags;
uint32_t index;
char *path;
mme_metadata_hdl_t *metadata;
} mme_explore_info_t;
```

Description:

The structure **mme_explore_info_t** carries information about items (folders and files) found at a specified path on a mediastore. It contains at least the members described in the table below.

Member	Type	Description
flags	**uint32_t**	Flags set to a value defined by MME_EXPLORE_* bit masks, described below.
index	**unint32_t**	Index for this entry in the parent folder.
path	**char**	A pointer to the full path to the item on the mediastore.
metadata	**mme_metadata_hdl_t**	A pointer to the metadata for this item, if metadata was requested and found. If this pointer is not zero, you know that metadata for this item is available. You do *not* need to check the MME_EXPLORE_FLAGS_HAS_METADATA flag as well.

MME_EXPLORE_* bit masks

Bitmasks that support mediastore exploration are described in the table below:

Constant	Value	Description
MME_EXPLORE_FILTER_INCLUDE	0x00000000	Inbound flag: instruct the MME to treat the file filter specification as an include-only specifier. This is the default setting if no flag is specified.
MME_EXPLORE_FLAGS_IS_FOLDER	0x00000001	The item is a folder — *not* a file.
MME_EXPLORE_FLAGS_IS_PLAYLIST	0x00000002	The item is a playlist (folder or file).
MME_EXPLORE_FLAGS_IS_PLAYLIST_ITEM	0x00000004	The item is a name from a playlist.
MME_EXPLORE_FLAGS_IS_PLAYLIST_FILENAME	0x00000008	The item is a resolved filename from a playlist file. The MME returns this value only for items retrieved from playlists when MME_EXPLORE_RESOLVE_PLAYLIST_ITEM is used for items successfully converted to a file on the mediastore. Otherwise, the MME returns the MME_EXPLORE_FLAGS_IS_PLAYLIST_ITEM flag with the item.
MME_EXPLORE_FLAGS_HAS_METADATA	0x00000100	The item has metadata.
MME_EXPLORE_RESOLVE_PLAYLIST_ITEM	0x00010000	Inbound flag: instruct the MME to resolve playlist file entries immediately. Using this flag results in much faster resolution of playlist contents to playable files, but the actual playlist entry value is not visible at to the client application.
MME_EXPLORE_FILTER_EXCLUDE	0x00020000	Inbound flag: instruct the MME to treat the file filter specification as an exclude specifier.

Below is an example from the command-line application **mmexplore** showing how MME_EXPLORE_* bit masks can be used.

```
static const char *item_type_str(uint32_t flags)
{
    if (flags & MME_EXPLORE_FLAGS_IS_PLAYLIST_FILENAME) {
        return "PF";
    }
    if (flags & MME_EXPLORE_FLAGS_IS_PLAYLIST_ITEM) {
        return "PI";
    }
    if ((flags & (MME_EXPLORE_FLAGS_IS_PLAYLIST|MME_EXPLORE_FLAGS_IS_FOLDER)) ==
        (MME_EXPLORE_FLAGS_IS_PLAYLIST|MME_EXPLORE_FLAGS_IS_FOLDER)) {
        return "DP";
    }
    if (flags & MME_EXPLORE_FLAGS_IS_PLAYLIST) {
```

```
        return "P ";
    }
    if (flags & MME_EXPLORE_FLAGS_IS_FOLDER) {
        return "D ";
    }
    return "F ";
}
```

Classification:

QNX Multimedia

See also:

mme_explore_end(), **mme_explore_hdl_t**, *mme_explore_info_free()*, *mme_explore_info_get()*, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_playlist_find_file()

Convert playlist file entries to filenames

Synopsis:

```
#include <mme/explore.h>

const mme_explore_info_t
          *mme_explore_playlist_find_file( mme_hdl_t *hdl,
                                           uint64_t msid,
                                           const char *entry,
                                           const char *path,
                                           const char *metadata_types,
                                           uint32_t flags );
```

Arguments:

hdl	A handle to the MME returned by *mme_explore_start()*.
msid	The ID of the media store to explore.
entry	The playlist file entry retrieved from the explorer.
path	The path of the playlist file on the mediastore.
metadata_types	An optional pointer to a string containing a comma-separated list of metadata types to retrieve. This pointer may be NULL. See METADATA_* in this reference.
flags	For future use.

Library:

mme

Description:

The function *mme_explore_playlist_find_file()* converts playlist file entries retrieved during exploration of a playlist file or folder to a filename on the system, and returns information about these converted entries in a **mme_explore_info_t** structure.

- You should convert your playlists to UTF-8 before calling *mme_explore_playlist_find_file()*. This function currently assumes that the *entry* argument is in UTF-8 character encoding. Characters in playlists may not be in UTF-8 encoding, and if they are not converted to UTF-8 may cause the function to fail.
- Since *mme_explore_playlist_find_file()* cannot know the origin of entries it converts, it always returns a value of 0 for the *index* member of the returned **mme_explore_info_t** structure.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

A populated **mme_explore_info_t** structure on success, or NULL if an error occurred (*errno* is set).

The result of a successful call to *mme_explore_playlist_find_file()* must be released by *mme_explore_info_free()*.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **mme_explore_hdl_t**, *mme_explore_info_free()*, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_position_set()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_position_set()

Position the information extraction location in the current folder

Synopsis:

```
#include <mme/explore.h>

int mme_explore_position_set( mme_explore_hdl_t *x_hdl,
                              unsigned offset,
                              unsigned items,
                              const char *metadata_types,
                              const char *filter,
                              uint32_t flags );
```

Arguments:

x_hdl	An explorer handle returned by *mme_explore_start()*.
offset	The offset in the folder from which to start getting information.
items	The number of items, starting at the offset from which information is required.
metadata_types	An optional pointer to a string containing a comma-separated list of metadata types to retrieve. This pointer may be NULL. See METADATA_* in this reference.
filter	A pointer to a regular expression used for filtering. This pointer may be NULL. See “Filtering” below.
flags	An MME_EXPLORE_FILTER_* bitmask instructing the MME to treat the filter specification as either an include-only or as an exclude-only specifier. The default is MME_EXPLORE_FILTER_INCLUDE. See “Filtering” below.

Library:

mme

Description:

The function *mme_explore_position_set()* sets:

- The position offset in the current folder from which the MME extracts information.
- The number of items that are requested, starting at the offset.
- The metadata types, if any, returned with the items. See also the chapter Metadata and Artwork in the *MME Developer’s Guide*.

If the item being explored is a playlist file, no metadata will be returned.

CAUTION: Retrieving more items than can be shown at one time in the HMI display window reduces system responsiveness:

- Always set *items* (the number of items requested) to a value less than or equal to the number of items that can be shown at one time in the HMI display window size.
- Adjust the number of items requested to correspond to changes to the size of the HMI display window.

Filtering

You can use the *filter* and *flag* arguments to filter the files examined and deliver only files of interest.

If the *filter* argument is NULL, it specifies no filter, and removes any previously used filter. When this argument is not NULL, it is an extended regular expression as defined by the *regcomp()* function, where the flags REG_ICASE **|** REG_EXTENDED **|** REG_NOSUB are used.

For example, to include only MP3 and WAVE files, based on the extensions **.mp3** and **.wav**, you should call *mme_explore_position_set()* as follows:

```
rc = mme_explore_position_set( x_hdl, 0, 20, NULL, ".mp3$|.wav$",
                               MME_EXPLORE_FILTER_INCLUDE );
```

Or, to exclude all files with the extension **.mov**, do the following:

```
rc = mme_explore_position_set( x_hdl, 0, 20, NULL, ".mov$",
                               MME_EXPLORE_FILTER_EXCLUDE );
```

CAUTION:

- The presence of filters makes using *mme_explore_size_get()* an expensive operation (for mediastores for which it is normally inexpensive), because the *mme_explore_size_get()* operation must now traverse the entire session to determine the actual number of items of interest.
- If a filter is assigned (or removed), the current position with the current explore session is reset to 0.
- If *mme_explore_size_get()* is called before the filter is set, its result may not be accurate when the filter is applied.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **`mme_explore_hdl_t`**, *mme_explore_info_free()*, *mme_explore_info_get()*, **`mme_explore_info_t`**, *mme_explore_playlist_find_file()*, *mme_explore_size_get()*, *mme_explore_start()*

mme_explore_size_get()

Get the number of entries to be explored

Synopsis:

```
#include <mme/explore.h>

ssize_t mme_explore_size_get( mme_explore_hdl_t *x_hdl,
                              uint32_t flags );
```

Arguments:

x_hdl An explorer handle returned by *mme_explore_start()*.

flags For future use.

Library:

mme

Description:

The function *mme_explore_size_get()* returns the number of entries of interest found in the folder that is currently being explored.

CAUTION:

- *mme_explore_size_get()* may require considerable time to complete execution: with some mediastore types, it requires a *readdir()* of the entire item being explored.
- If *mme_explore_size_get()* is called before the filter is set, its result may not be accurate when the filter is applied.
- The use of filters with *mme_explore_position_set()* makes using *mme_explore_size_get()* an expensive operation (for mediastores for which it is normally inexpensive), because the *mme_explore_size_get()* operation must now traverse the entire session to determine the actual number of items of interest.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **mme_explore_hdl_t**, *mme_explore_info_free()*, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_start()*

mme_explore_start()

Start exploring an item on a mediastore

Synopsis:

```
#include <mme/explore.h>

mme_explore_hdl_t *mme_explore_start( mme_hdl_t *hdl,
                                      uint64_t uint64_t msid,
                                      const char *path,
                                      uint32_t flags );
```

Arguments:

hdl An MME connection handle.

msid The ID of the media store to explore.

path The path to the item to explore. Use an empty string to start at the root of the media store. See the "Description" below.

flags For future use.

Library:

mme

Description:

The function *mme_explore_start()* returns a handle to be used to explore a mediastore. After calling *mme_explore_start()*, you can use other *mme_explore_*()* functions to find and learn about folders and files of interest on the media store.

The *path* argument can be refer to a file marked as a playlist as well as to a folder or to a file that can be played.

Events

None delivered.

Blocking and validation

This function performs no validations. It doesn't block.

Returns:

An initialized **mme_explore_hdl_t**, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_explore_end(), **mme_explore_hdl_t**, *mme_explore_info_free()*, *mme_explore_info_get()*, **mme_explore_info_t**, *mme_explore_playlist_find_file()*, *mme_explore_position_set()*, *mme_explore_size_get()*

Synopsis:

```
#include <mme/interface.h>

#define MME_PLAYMODE_*

#define MME_FORMAT_*
```

Description:

The constants MME_PLAYMODE_* define the play mode used for a track session. The constants MME_FORMAT_* described in the table below define the media file formats (codecs). They are used by the *format* field in the **`library`** table.

Constant	Value	Description
MME_PLAYMODE_LIBRARY	0x0	Library mode.
MME_PLAYMODE_FILE	0x2	File-based track session mode.
MME_FORMAT_UNKNOWN	0ULL	Unknown media format.
MME_FORMAT_MLP	1ULL	Meridian Lossless Packing
MME_FORMAT_PCM	2ULL	LPCM and PCM (Pulse-Code Modulation)
MME_FORMAT_AC3	3ULL	AC-3 (Dolby Digital)
MME_FORMAT_MP2	4ULL	MPEG1 audio layer II
MME_FORMAT_MPEG1_L2	4ULL	MPEG audio layer II
MME_FORMAT_DTS	5ULL	DTS Coherent Acoustics (Digital Theatre Systems)
MME_FORMAT_SDDS	6ULL	Sony Dynamic Digital Sound
MME_FORMAT_MPEG1_L1	7ULL	MPEG1 audio layer I
MME_FORMAT_MPEG1_L3	8ULL	MPEG1 audio layer III
MME_FORMAT_MPEG2_L1	9ULL	MPEG1 audio layer I
MME_FORMAT_MPEG2_L2	10ULL	MPEG2 audio layer II
MME_FORMAT_MPEG2_L3	11ULL	MPEG2 audio layer III
MME_FORMAT_MPEG2_PRO	12ULL	MPEG2 program stream
MME_FORMAT_OGG	13ULL	Ogg Vorbis format
MME_FORMAT_AAC	14ULL	AAC format

continued...

Constant	Value	Description
MME_FORMAT_AMR	15ULL	AMR format
MME_FORMAT_PCM_PREEMPH	16ULL	PCM format with pre-emphasis
MME_FORMAT_WMA	17ULL	WMA format

Classification:

QNX Multimedia

See also:

MME_MSCAP_*, MME_MSCAP_*, MME_STORAGETYPE_*, MME_SYNC_OPTION_*, **mediastores**

mme_get_api_timeout_remaining()

Get the time left on the unblocking timer

Synopsis:

```
#include <mme/mme.h>

int mme_get_api_timeout_remaining( mme_hdl_t *hdl,
                                   uint32_t *milliseconds );
```

Arguments:

hdl	An MME connection handle.
milliseconds	Deprecated.

Library:

mme

Description:

The function *mme_get_api_timeout_remaining()* distinguishes between EINTR errors caused by the MME unblocking the caller and other EINTR errors.

If a client application has used *mme_set_api_timeout()* to set an unblocking timer on the control context, API calls that are blocked beyond the set timeout period will unblock the client, returning early with the *errno* set to EINTR.

Because *errno*s propagate up, an EINTR can be returned to the client for reasons other than a timeout. To distinguish EINTR errors caused by the MME unblocking the caller and other EINTR errors, call *mme_get_api_timeout_remaining()* to get the time remaining on the timer. If the time remaining indicated by *milliseconds* is greater that 0 (zero), then the EINTR error wasn't caused by a timeout. If the time remaining is 0, then the EINTR was caused by a timeout.

The MME's default configuration is to disable unblocking capabilities, which renders the information delivered by *mme_get_api_timeout_remaining()* meaningless. To enable the MME's unblocking capability, set the **<Unblock>** configuration element attribute to "true".

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥0 Success. Assuming an MME EINTR

-1 An error occurred (*errno* is set). Errno is set. An EINVAL error indicates that the timeout is set to 0, so the request for the time remaining is invalid.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_set_api_timeout()

mme_get_event()

Get the next queued MME event

Synopsis:

```
#include <mme/mme.h>

int mme_get_event( mme_hdl_t *hdl,
                   mme_event_t **mme_event );
```

Arguments:

hdl	An MME connection handle.
mme_event	A pointer to a pointer to the event in the MME event queue.

Library:

mme

Description:

The function *mme_get_event()* allows you to determine when your client application receives events. It retrieves events from the event queue, and places event information in the **mme_event_t** data structure. This information includes the event:

- type
- size, in bytes (events are variable length)
- data

Events are associated with an MME connection handle **mme_hdl_t**; they cannot be cleared by the client application.

The MME does not automatically place events in the event queue. You must use the function *mme_register_for_events()* to register for the types of events your client application needs to receive. Registration is typically done immediately after connection.

When the client application is registered for one or more type of event, the MME places these event types in an event queue and sends the relevant **sigevent** to the client application. Based on the **sigevent**, the client can decide to call *mme_get_event()* to retrieve the event.

A call to *mme_get_event()* invalidates any data that was in the **mme_event_t** before the call was made. If the client application needs to keep event information longer than the next call to *mme_get_event()*, it must copy the event before calling *mme_get_event()*.

For more information about registering for events, see "Registering for events" in the chapter Starting Up and Connecting to the MME of the *MME Developer's Guide*, and *mme_register_for_events()*.

If your client application does not register for events before it calls *mme_get_event()*, the event queue will be empty. If there are no events in the event queue **`mme_event_type_t`** will be set to MME_EVENT_NONE.

For more information about these data structures, see the relevant sections in the chapter MME Events.

Events

None delivered.

Blocking and validation

This function doesn't perform any validations, and blocks only on internal event structures. It doesn't block on processes external to the MME, such as **`qdb`** or **`io-media`**.

Returns:

≥**`0`**	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect(), *mme_disconnect()*, *mme_register_for_events()*, MME Events, "Registering for events" in the *MME Developer's Guide*

mme_get_logging()

Get the verbosity log levels for specified logging modules

Synopsis:

```
#include <mme/mme.h>

int mme_get_logging( mme_hdl_t *hdl,
                     const char *name,
                     char *settings,
                     size_t size );
```

Arguments:

hdl	An MME connection handle.
name	A pointer to a string with the name of the logging module for which information is required. Set the string to NULL to retrieve information for all logging modules.
settings	A comma-separated list of the logging modules and their log levels. See "Log level settings" below.
size	The size, in bytes, of the buffer for the retrieved setting information.

Library:

mme

Description:

The function *mme_get_logging()* retrieves the logging verbosity levels for the specified MME logging modules. For more information about the logging modules and how to set their levels, see *mme_set_logging()*.

Log level settings

The *mme_get_logging()* function writes logging level information into the buffer referenced by the *settings* argument. These settings are written as a comma-separated list with each item based on the following template:

module=verbosity level:flags

For example, if the metadata interface logging module has a verbosity level of 8 and its flags set to 0, *mme_get_logging()* writes the following to the buffer referenced by *settings*: **mdi=8:0**.

Logging modules

The strings that identify **mme** logging modules include:

String	Module
`imgprc`	image processing module
`mdi`	metadata interface module
`mdp`	metdata plugin module
`pl`	playlist module
`sync`	synchronization module
`mme`	all other modules

The above list is not definitive. The logging modules may change. To find out what logging module strings are valid, call *mme_get_logging()* with the string referenced by the *name* argument set to NULL.

Logging flags

The logging flags are bit masks that configure logging behavior:

Value	Behavior
1	Also write anything logged to standard output.
2	Write timing logs.

Events

None delivered.

Blocking and validation

This function doesn't perform any validations, and doesn't block.

Returns:

`≥0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_set_logging()

Synopsis:

```
#include <mme/mme.h>

int mme_get_title_chapter( mme_hdl_t hdl,
                           uint64_t *title,
                           uint64_t *ntitles,
                           uint64_t *chapter,
                           uint64_t *nchapters );
```

Arguments:

hdl	An MME connection handle.
title	The current title number.
ntitles	The number of titles in currently playing track or mediastore.
chapter	The current chapter number.
nchapters	The number of chapters in the current title.

Library:

mme

Description:

The function *mme_get_title_chapter()* gets for the currently playing DVD track:

- the number of titles and chapters on the track or its mediastore
- the currently playing title and chapter numbers.

This function can be used only if the MME_PLAYSUPPORT_NAVIGATION flag is set in the *support* member of the structure **mme_play_info_t**.

To start playback from a specific title and chapter, call the function *mme_seek_title_chapter()* to seek to the desired title and chapter, then call the function *mme_play()* to start playback.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Examples:

Below is a code snippet that illustrates how to get DVD title and chapter information.

```
uint64_t title, ntitles, chapter, nchapters;

rc = mme_get_title_chapter( mmehdl, &title, &ntitles, &chapter, &nchapters);
if (rc == EOK) {
    printf( "Title %lld of %lld, Chapter %lld of %lld",
        title, ntitles, chapter, nchapters);
} else {
    printf( "Error getting title/chapter info %s", strerror(errno));
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), *mme_play_bookmark()*, *mme_play_get_info()*, **mme_play_info_t**, *mme_seek_title_chapter()*, *mme_seektotime()*

mme_getautopause()

Get the autopause mode set for a control context

Synopsis:

```
#include <mme/mme.h>

int mme_getautopause( mme_hdl_t *hdl);
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_getautopause()* returns the autopause mode for a control context. It returns 1 if autopause is enabled, 0 if it isn't enabled. For a description of autopause mode, see *mme_setautopause()*.

Events

None delivered

Blocking and validation

This function doesn't block.

Returns:

≥0 Success:

- **1** Autopause mode is set.
- **0** Autopause mode is not set.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_next(), mme_play(), mme_prev(), mme_setautopause()

Get the control context ID for the currently connected control context

Synopsis:

```
#include <mme/mme.h>

int mme_getccid( mme_hdl_t *hdl,
                 uint64_t *ccid);
```

Arguments:

hdl An MME connection handle.

ccid The control context ID (output).

Library:

mme

Description:

The function *mme_getccid()* returns the ID for the control context associated with the specified MME handle. You can use this ID to query these tables in the MME database:

- **controlcontext**, to obtain additional information about the control context (such as its current track session)
- **nowplaying**, to find the metadata for the track currently playing on the control context.

For more information about control contexts, see the chapter Control Contexts, Zones and Output Devices in the *MME Developer's Guide*.

Events

None delivered.

Blocking and validation

This function is fully validating; it checks all arguments before returning.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect()

mme_getclientcount()

Get the number of clients connected to a control context

Synopsis:

```
#include <mme/mme.h>

int mme_getclientcount( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_getclientcount()* returns the number of clients connected to the MME on the control context specified by *hdl*. This count is the number of **mme_hdl_t** active handles that have been returned by calls to *mme_connect()* for the control context.

Events

None delivered.

Blocking and validation

This function is non-blocking and performs no validations.

Returns:

≥0 Success: the number of clients attached to the control context for the specified MME handle.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect()

Synopsis:

```
#include <mme/mme.h>

int mme_getlocale( mme_hdl_t *hdl,
                   char *locale );
```

Arguments:

hdl An MME connection handle.

locale A pointer to a location where the function can store the current locale setting. This location must be at least six characters long.

Library:

mme

Description:

The function *mme_getlocale()* gets the current locale setting for an MME control context.

Events

None delivered

Blocking and validation

This function doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_setlocale()

Synopsis:

```
#include <mme/mme.h>

int mme_getrandom( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_getrandom()* tells you whether the specified control context has been set to random playback mode. On success, it returns the control context's random mode.

See **mme_mode_random_t** for a description of the random modes.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns. Verifies that the client application code is valid. Blocks on control contexts.

If *mme_getrandom()* is called and another function is called before *mme_getrandom()* returns, the second function blocks on **io-media** until *mme_getrandom()* returns. If there are no other pending calls, *mme_getrandom()* returns without blocking on **io-media**.

Returns:

≥**0** Success: the random playback mode for the control context.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getrepeat(), mme_getscanmode() mme_setrandom() mme_setrepeat(), **`mme_mode_random_t`**, **`mme_mode_repeat_t`**

Synopsis:

```
#include <mme/mme.h>

int mme_getrepeat( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_getrepeat()* returns the repeat mode for the specified control context. On success, it returns the control context's repeat mode.

See **mme_mode_repeat_t** for a description of these modes.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

This function blocks on control contexts. If *mme_getrepeat()* is called and another function is called before *mme_getrepeat()* returns, the second function blocks on **io-media** until *mme_getrepeat()* returns. If there are no other pending calls, *mme_getrepeat()* returns without blocking on **io-media**.

Returns:

≥0 Success: the repeat playback mode for the control context.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No

continued...

Safety	
Thread	Yes

See also:

mme_getrandom(), mme_getscanmode() mme_setrandom() mme_setrepeat(), **`mme_mode_random_t`**, **`mme_mode_repeat_t`**

Synopsis:

```
#include <mme/mme.h>

int mme_getscanmode( mme_hdl_t *hdl,
                     uint64_t *time );
```

Arguments:

hdl An MME connection handle.

time A pointer to a location where the function can store the scan mode setting (in milliseconds).

Library:

mme

Description:

The function *mme_getscanmode()* gets the scan mode setting for a control context. This setting is the number of milliseconds of a track that the MME plays in scan mode before skipping to the next track in the tracklist.

Events

None delivered.

Blocking and validation

This function blocks on control contexts. If *mme_getscanmode()* is called and another function is called before *mme_getscanmode()* returns, the second function blocks on **io-media** until *mme_getscanmode()* returns. If there are no other pending calls, *mme_getscanmode()* returns without blocking on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getrandom(), mme_getrepeat(), mme_setrandom() mme_setrepeat(), mme_setscanmode()

Synopsis:

```
#include <mme/types.h>
```

Description:

The opaque structure **mme_hdl_t** carries MME connection handle information. Valid connection handles are created by the function *mme_connect()*. The MME fills in all needed information to create the connection handle; you only need to know that all calls to MME functions require a valid connection handle.

The function *mme_disconnect()* releases connection handles. Function calls made with a connection handle after it has been released will cause an error.

Safety

All MME functions are thread-safe. The client application can create multiple connections and the MME handles thread safety for all threads *when each thread uses a different connection handle*.

However, if you use the same connection handle for more than one thread in your client application, you must use mutexes, semaphores or some other method to protect the connection handle from being accidently overwritten.

Classification:

QNX Multimedia

See also:

mme_connect(), *mme_disconnect()*

mme_lib_column_set()

Set values in specified table column

Synopsis:

```
#include <mme/mme.h>

int mme_lib_column_set( mme_hdl_t *hdl,
                        uint64_t msid,
                        const char *column,
                        int value );
```

Arguments:

hdl	An MME connection handle.
msid	The ID of the mediastore for which a value needs to be changed in the **library** table.
column	The name of the **library** table column that needs to be changed.
value	The new value for the entries for the specified mediastore in the specified column.

Library:

mme

Description:

The function *mme_lib_column_set()* inserts a value into the entries for a mediastore in the **library** table (or adjunct tables). It can be used to perform actions such as clearing the **library** table *accurate* fields for the specified mediastore.

Update behavior

This function can only be used to update entries in the columns listed below, and it validates that the character string referenced by *column* specifies one of these columns:

- *accurate*
- *last_played*
- *fullplay_count*
- *playable*
- *permanent*
- *copied_fid*

- When *mme_lib_column_set()* completes execution it returns the number of rows for the specified mediastore that now have the new value. In other words, the function returns the number of rows for the specified mediastore that are now set to the new value.
- If prior to the call to *mme_lib_column_set()* some rows were already set to the required value, the return value may differ from the number of rows actually *updated*.
- *Only* rows for the specified mediastore are included in the return value. Rows for other mediastores are not counted.

Events

None delivered.

Blocking and validation

This function validates the column name; it executes to completion.

Returns:

≥0 Success: the number of table rows for the specified mediastore, with the new value updated. See “Update behavior” above.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), mme_resync_mediastore(), mme_setpriorityfolder(), mme_sync_cancel(), mme_sync_directed(), mme_sync_file(), mme_sync_get_msid_status(), mme_sync_get_status()

mme_media_get_def_lang()

Get the preferred media playback language

Synopsis:

```
#include <mme/mme.h>

int mme_media_get_def_lang ( mme_hdl_t *hdl,
                             char *lang );
```

Arguments:

hdl An MME connection handle.

lang A pointer to a location where the function can store the current preferred media playback language (a string to place a 0-terminated, 2-character ISO639-1 language code). If the language hasn't been set, *lang* is set to a 0-length string.

Library:

mme

Description:

The function *mme_media_get_def_lang()* gets the current preferred language playback setting for an MME control context.

For more information about default language settings, see *mme_media_set_def_lang()*.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_media_set_def_lang()

mme_media_set_def_lang()

Set the preferred media playback language

Synopsis:

```
#include <mme/mme.h>

int mme_media_set_def_lang ( mme_hdl_t *hdl,
                             const char *lang );
```

Arguments:

hdl An MME connection handle.

lang The default language code to set. This is a string containing 2-character ISO639-1 language code. See **http://www.loc.gov/standards/iso639-2/php/code_list.php**

Library:

mme

Description:

The function *mme_media_set_def_lang()* sets the preferred language for media playback. After this function sets the language preference for the current MME control context, the MME uses the selected language as the default langauge for playback whenever possible. For example, *mme_media_set_def_lang()* sets the preferred language to German:

- If a DVD-video has playback in German, the MME will play the DVD in German.
- If a DVD-video does not have playback in German, the MME will play the DVD in the preferred language set on the DVD itself.

If *mme_media_set_def_lang()* is not called after connecting to the MME, no language preference is selected, and the MME will play media in the preferred language set on the mediastores.

Events

The function *mme_media_set_def_lang()* delivers the MME_EVENT_DEFAULT_LANGUAGE so that asynchronous clients are notified that the default preferred language has been successfully set, or that the attempt to change the language has failed.

Blocking and validation

This function doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_media_get_def_lang()

mme_mediacopier_add()

Add files to the copy queue

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_add( mme_hdl_t *hdl,
                         mme_mediacopier_info_t *copyinfo,
                         char *statement,
                         uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
copyinfo	A pointer to a **mme_mediacopier_info_t** structure that contains information about the copy operation.
statement	An SQL statement that selects the fids that you want to encode.
flags	Flags affecting the copy operation. See “Mediacopier flags” below.

Library:

mme

Description:

The function *mme_mediacopier_add()* prepares a media copying or ripping operation. Files that are selected by *statement* are added to the **copyqueue** table in the MME database.

To start a copy or ripping operation:

- use *mme_mediacopier_add()* to populate the **copyqueue** table with information needed for the copy or ripping operation
- call *mme_mediacopier_enable()* to start the operation

To add files to the copy queue, specifying strings for unknown metadata, use *mme_mediacopier_add_with_metadata()*.

Using default ripping values

By default, if you set the *copyinfo* members as follows: **dstmsid=0**, **dstfolder=NULL**, **dstfilename=NULL**, and **encodeformatid=0**, the MME will use the defaults in the configuration file **mme.conf**.

CAUTION: You should not assume that the default destination mediastore set by the configuration element **<Copying>**/**<Destination>**/**<MSID>** is always your HDD. In some instances, on startup the MME may detect another mediastore, such as a CD, *before* it detects the HDD and assign it *msid*=1. When preparing a media copy or ripping operation, ensure that the destination mediastore (*dstmsid*) is a writeable mediastore.

For more information, see the chapter Configuring Media Copying and Ripping in the *MME Configuration Guide*.

Mediacopier flags

Media copying and ripping uses the *flags* argument to determine media copying and ripping behavior. Possible values are:

- MME_MEDIACOPIER_COPYADD_NONE — copy or rip directly to a destination folder
- MME_MEDIACOPIER_COPYADD_PRESERVE_PATH — preserve the original folder structure for copied or ripped files. Create folders as required.
- MME_MEDIACOPIER_USE_DEFAULT_FILENAME — use default destination filename set in the MME configuration file. See “Configuration elements for the media copy and ripping destination” in the Configuring Media Copying and Ripping chapter of the *MME Configuration Guide*.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0** Success

-1 An error occurred (*errno* is set).

Examples:

Below is a code snippet from the **mmecli.c** example application. This code snippet illustrates how to set up a call to *mme_mediacopier_add()*.

```
mme_mediacopier_info_t copyinfo;

 // Just use defaults for now
 copyinfo.dstmsid = 0;
 copyinfo.dstfolderid = 0;
 copyinfo.format = 0;
 copyinfo.bitrate = 0;
```

```
rc = mme_mediacopier_add(&mmehdl, &copyinfo, statement, 0);

if (rc == -1) {
   sprintf(output, "Error setting copy add");
}
else {
   sprintf(output, "copy added");
}
```

The example below shows how to use template strings for the destination folder and file name.

```
mme_mediacopier_info_t copyinfo;

char *folder = "/ripped/$ARTIST/$ALBUM/";
char *title = "$0TRACK-$TITLE(date=$DATESTAMP,time=$TIMESTAMP,srcfid=$SR

copyinfo.dstmsid = 1;
copyinfo.dstfolder = folder;
copyinfo.dstfilename = title;
copyinfo.encodeformatid = 2;
rc = mme_mediacopier_add(mmehdl, &copyinfo, statement, 0);

 if (rc == -1) {
    sprintf(output, "Error setting copy add");
 }
 else {
    sprintf(output, "copy added");
 }
```

See **mme_mediacopier_info_t**.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add_with_metadata(), *mme_mediacopier_cleanup()*, *mme_mediacopier_clear()*, *mme_mediacopier_disable()*, *mme_mediacopier_enable()*, *mme_mediacopier_get_status()*, *mme_mediacopier_remove()*, **mme_mediacopier_info_t**

mme_mediacopier_add_with_metadata()

Add files to the copy queue, specifying strings for unknown metadata

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_add_with_metadata( mme_hdl_t *hdl,
                                       mme_mediacopier_info_t *copyinfo,
                                       const char *statement,
                                       uint32_t flags,
                                       const char *unknown_album,
                                       const char *unknown_artist );
```

Arguments:

hdl	An MME connection handle.
copyinfo	A pointer to a **mme_mediacopier_info_t** structure that contains information about the copy operation.
statement	An SQL statement that selects the fids that you want to encode.
flags	Flags affecting the copy operation.
unknown_album	A pointer to a text string to add to the file metadata if the album is not known.
unknown_artist	A pointer to a text string to add to the file metadata if the artist is not known.

Library:

mme

Description:

The function *mme_mediacopier_add_with_metadata()* prepares a media copying or ripping operation and adds specified strings when the artist or album is not known. This function behaves exactly like *mme_mediacopier_add()*, except for the added functionality required to add the string for unknown metadata.

This function updates metadata if:

- the **<IgnoreNonAccurate>** configuration element is set to **true**, and the source track **library.accurate** value is 0 (the accuracy of metadata is not know); in this case, the function uses the default file and folder metadata;

  or, if:

- the *flags* argument is set to MME_MEDIACOPIER_USE_METADATA

mme_mediacopier_add_with_metadata() updates the metadata both in the MME **library** table entry for the destination file, and in the destination file itself. This

behavior ensures that the metadata added to the destination file is maintained, even in the event that the MME database is lost.

To use *mme_mediacopier_add_with_metadata()* to specify metadata for destination files for which the album or artist is not known:

1. Specify values for the strings that will complete the *unknown_album* and *unknown_artist* fields for the destination files. Include the **$MSIDENTIFIER** environment variable in the strings to ensure that each file is uniquely identified. See **$MSIDENTIFIER** below.
2. Call *mme_mediacopier_add_with_metadata()*.

$MSIDENTIFIER

The **$MSIDENTIFIER** environment variable is set to the value of the *identifier* field in the **`mediastores`** table. Adding it to the string written into a destination file's *unknown_** fields ensures that the destinatation file is always correctly associated with its mediastore.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥0. Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_mediacopier_add()*, *mme_mediacopier_cleanup()*, *mme_mediacopier_clear()*, *mme_mediacopier_disable()*,

mme_mediacopier_enable(), *mme_mediacopier_get_status()*, *mme_mediacopier_remove()*, **`mme_mediacopier_info_t`**

mme_mediacopier_cleanup()

Clean up partially copied or ripped files

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_cleanup( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_mediacopier_cleanup()* cleans up partially copied or ripped files from the MME database and the system HDD. You should use this function when starting up after a media copying or ripping operation has been aborted or was stopped unexpectedly, in order to to ensure that the MME does not keep entries for incompletely ripped files in its database.

The function *mme_mediacopier_cleanup()* can be called only if the mediacopier is disabled. An attempt to call this function while the mediacopier is enabled causes it to return an EBUSY error.

Events

None delivered.

Blocking and validation

This function checks that the mediacopier is disabled; it doesn't block.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set). An EBUSY error indicates that the mediacopier is enabled.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), *mme_mediacopier_clear()*, *mme_mediacopier_disable()*, *mme_mediacopier_enable()*, *mme_mediacopier_get_status()*, *mme_mediacopier_remove()*

mme_mediacopier_clear()

Clear all files from the media copy queue

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_clear( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_mediacopier_clear()* removes all files from the media copy queue.

To remove specific files from the copy queue, use *mme_mediacopier_remove()*.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), mme_mediacopier_cleanup(), mme_mediacopier_disable(), mme_mediacopier_enable(), mme_mediacopier_get_status(), mme_mediacopier_remove()

mme_mediacopier_disable()

Disable the mediacopier

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_disable( mme_hdl_t *hdl,
                             uint32_t flags );
```

Arguments:

hdl The MME connection handle.

flags Flags that affect the disable operation. None are defined; pass as 0.

Library:

mme

Description:

The function *mme_mediacopier_disable()* stops a copying or ripping operation.

Stopping a media copying or ripping operation does not affect the **copyqueue** table. To remove file from **copyqueue** table, you must call the function *mme_mediacopier_clear()*.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), *mme_mediacopier_cleanup()*, *mme_mediacopier_clear()*, *mme_mediacopier_enable()*, *mme_mediacopier_get_status()*, *mme_mediacopier_remove()*

mme_mediacopier_enable()

Enable the mediacopier

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_enable( mme_hdl_t *hdl,
                            uint32_t flags );
```

Arguments:

hdl The MME connection handle.

flags Flags that affect the enable operation. None are defined; pass as 0.

Library:

mme

Description:

The function *mme_mediacopier_enable()* starts a copying or ripping operation.

Before calling *mme_mediacopier_enable()* you must call *mme_mediacopier_add()* to prepare a media copy operation and populate the **copyqueue** table. You can stop a copy operation in progress by calling *mme_mediacopier_disable()*.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), mme_mediacopier_cleanup(), mme_mediacopier_clear(), mme_mediacopier_disable(), mme_mediacopier_get_status(), mme_mediacopier_remove()

mme_mediacopier_get_mode()

Get the selected media copy or rip mode

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_get_mode( mme_hdl_t *hdl,
                              mme_mediacopier_mode_t *copymode );
```

Arguments:

hdl	An MME connection handle.
copymode	The copy mode selected for the media copying or ripping operation.

Library:

mme

Description:

The function *mme_mediacopier_get_mode()* gets the selected mode for a media copy or ripping operation. This mode is defined by the enumerated type **mme_mediacopier_mode_t**.

Events

None delivered.

Blocking and validation

This function blocks until it completes.

Returns:

≥**0**	Success
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_cleanup(), mme_mediacopier_set_mode(), mme_metadata_set()

mme_mediacopier_get_status()

Get the status of a media copy or ripping operation

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_get_status( mme_hdl_t *hdl,
                                mme_copy_status_t *copy_status);
```

Arguments:

hdl An MME connection handle.

msg A pointer to the structure **mme_copy_status_t** that is filled in by the function. See **mme_copy_status_t** below.

Library:

mme

Description:

The function *mme_mediacopier_get_status()* gets the status of a media copying or ripping operation. The status information is placed in a structure **mme_copy_status_t**.

mme_copy_status_t

```
typedef struct _mme_copy_status {
   uint64_t           cqid;
   uint64_t           srcfid;
   uint64_t           dstfid;
   uint32_t           units;
   uint32_t           reserved;
   union {
      mme_time_t      time_info;
      mme_byte_status_t byte_info;
   };
} mme_copy_status_t;
```

The structure **mme_copy_status_t** defines information about the current media copy or ripping operation. Its members include at least those described in the table below.

Member	Type	Description
cqid	**uint64_t**	The copy queue ID entry currently being copied or ripped.

continued...

Member	Type	Description
srcfid	**uint64_t**	The file ID of the source file being copied or ripped.
dstfid	**uint64_t**	The file ID of the destination file.
units	**uint32_t**	The units (time or bytes) used to track progress of the media copy or ripping operation. See **mme_copy_units_t** below.
reserved	**uint32_t**	Reserved for internal use.
byte_info **|** *time_info*	**union**	Depending on the value of *units*, either the structure **mme_time_t** with the play time ripped, or the structure **mme_byte_status_t** with the number of bytes copied.

mme_copy_units_t

The enumerated type **mme_copy_units_t** defines the units used to measure progress during a media copy or ripping operation. It can have the following values:

- MME_COPY_UNITS_NONE (0) — no measurement units have been defined.
- MME_COPY_UNITS_TIME_MS (1) — time, in milliseconds.
- MME_COPY_UNITS_BYTES (2) — bytes.

mme_byte_status_t

```
typedef struct _mme_byte_status {
    uint64_t    bytepos;
    uint64_t    nbytes;
} mme_byte_status_t;
```

Media copy operations use *byte_info* to communicate the progress of a copy operation when **mme_copy_units_t** is set to MME_COPY_UNITS_BYTES. *byte_info* is a member of **mme_copy_status_t**; it uses the structure **mme_byte_status_t** to hold the copy progress information. Its members are described in the table below.

Member	Type	Description
bytepos	**uint64_t**	Number of bytes copied thus far.
nbytes	**uint64_t**	Total number of bytes to be copied.

time_info

Ripping operations use *time_info* to communicate the progress of a ripping operation when **mme_copy_units_t** is set to MME_COPY_UNITS_TIME_MS. A member of **mme_copy_status_t**, *time_info* uses the structure **mme_time_t** to hold the ripping progress information, in milliseconds:

- the duration of the track
- the current time position

See **mme_time_t**.

Events

None delivered.

Blocking and validation

This function blocks until it completes.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), *mme_mediacopier_cleanup()*, *mme_mediacopier_clear()*, *mme_mediacopier_disable()*, *mme_mediacopier_enable()*, *mme_mediacopier_remove()*, **mme_time_t**

mme_mediacopier_info_t

Media copy and ripping information

Synopsis:

```
#include <mme/types.h>

typedef struct {
   uint64_t     dstmsid;
   const char   *dstfolder;
   const char   *dstfilename;
   uint64_t     encodeformatid;
} mme_mediacopier_info_t;
```

Description:

The structure **mme_mediacopier_info_t** carries information about a media copy or ripping operation. It includes at least the members described in the table below.

Member	Type	Description
dstmsid	**uint64_t**	The destination *msid*. Set to 0 to use the default *msid*.
dstfolder	**char**	A pointer to the destination folder for the tracks to be ripped. See *dstfolder* below.
dstfilename	**char**	A pointer to the string used to create the destination file name for the tracks to be ripped.
encodeformatid	**uint64_t**	The encode format ID (*encodeformatid*) from the **encodeformats** table that you want to use for encoding. See *encodeformatid* below.

dstfolder and *dstfilename*

The value for *dstfolder* must be in the format **/***foldername***/** (beginning and ending with a "/" character). For example, if in your **mediastores** table the destination *msid* has a mountpath of **/media/drive**, and the *dstfolder* name is "/ripped/", then the track is ripped to **/media/drive/ripped/**.

Set *dstfolder* to NULL to use the default destination folder, and *dstfilename* NULL to use the destination file name defined in the MME configuration file **mme.conf**. You can specify nested sub-directories, as required.

Destination folder *dstfolder* and file name *dstfilename* template strings

The MME defines templates strings you can use to name the ripping destination folders and files. These template strings are described in the table below.

String	Value	Description
$TITLE	MME_MEDIACOPIER_TEMPLATE_TITLE	song title
$ARTIST	MME_MEDIACOPIER_TEMPLATE_ARTIST	artist name
$ALBUM	MME_MEDIACOPIER_TEMPLATE_ALBUM	album name
$GENRE	MME_MEDIACOPIER_TEMPLATE_GENRE	song genre
$COMPOSER	MME_MEDIACOPIER_TEMPLATE_COMPOSER	song composer
$TRACK	MME_MEDIACOPIER_TEMPLATE_TRACK	track number
$0TRACK	MME_MEDIACOPIER_TEMPLATE_0TRACK	track number with leading zeros: 01, 02, etc.
$DISC	MME_MEDIACOPIER_TEMPLATE_DISC	disc number
$0DISC	MME_MEDIACOPIER_TEMPLATE_0DISC	disc number with leading zeros: 01, 02, etc.
$YEAR	MME_MEDIACOPIER_TEMPLATE_YEAR	release year
$SRCFID	MME_MEDIACOPIER_TEMPLATE_SRCFID	source file ID
$SRCMSID	MME_MEDIACOPIER_TEMPLATE_SRCMSID	source mediastore ID
$TIMESTAMP	MME_MEDIACOPIER_TEMPLATE_TIMESTAMP	time when file is copied
$DATESTAMP	MME_MEDIACOPIER_TEMPLATE_DATESTAMP	date when file is copied
$MSIDENTIFIER	MME_MEDIACOPIER_TEMPLATE_MSIDENTIFIER	source mediastore ID
$NO_PRESERVE_PATH	COPY_NO_PATH_PRESERVE	force the path to be discarded
$PRESERVE_PATH	COPY_PATH_PRESERVE	force the path to be preserved
$PRESERVE_PATH_AFTER	COPY_PATH_PRESERVE_AFTER	modify the source path when it is appended to the destination folder

encodeformatid

The standard default values for *encodeformatid* are:

- 1 — copy operation
- 2 — wav encoding
- 3 — AAC encoding (SH4 only; requires specific licences)
- 4 — wma encoding (requires specific licences)

Set *encodeformatid* to 0 to use the default encode format.

Classification:

QNX Multimedia

See also:

mme_mediacopier_add()

mme_mediacopier_remove()

Remove files from the media copy queue

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_remove( mme_hdl_t *hdl,
                            char *statement,
                            uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
statement	An SQL statement of copy queue IDs that you want to remove from the copy queue.
flags	Option flags. There are currently none defined, pass as 0.

Library:

mme

Description:

The function *mme_mediacopier_remove()* removes specified files from the copy queue. To clear all files from the copy queue, use *mme_mediacopier_clear()*.

Events

None returned.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), *mme_mediacopier_cleanup()*, *mme_mediacopier_clear()*, *mme_mediacopier_disable()*, *mme_mediacopier_enable()*, *mme_mediacopier_get_status()*

mme_mediacopier_set_mode()

Get the selected media copy or rip mode

Synopsis:

```
#include <mme/mme.h>

int mme_mediacopier_set_mode( mme_hdl_t *hdl,
                              mme_mediacopier_mode_t *copymode );
```

Arguments:

hdl	An MME connection handle.
copymode	The copy mode selected for the media copying or ripping operation.

Library:

mme

Description:

The function *mme_mediacopier_set_mode()* sets the mode for a media copying or ripping operation. This mode is defined by the enumerated type **mme_mediacopier_mode_t**.

mme_mediacopier_mode_t

The enumerated type **mme_mediacopier_mode_t** sets the media copying or ripping mode:

- MME_MEDIACOPIER_MODE_BKG — The MME will:
  - return after it initiates the operation
  - perform the media copy or ripping in the background
  - give priority to other operations
- MME_MEDIACOPIER_MODE_PRIORITY_BKG — The MME will:
  - return after it initiates the operation
  - perform the media copy or ripping in the background
  - take priority over other background operations
- MME_MEDIACOPIER_MODE_FOREGROUND (For future implementation.) — The MME will:
  - return after it completes the operation
  - perform the media copy or ripping in the foreground
  - negotiate priority with other foreground operations

Events

None delivered.

Blocking and validation

This function blocks until it completes.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_cleanup(), *mme_mediacopier_get_mode()*, *mme_metadata_set()*

mme_metadata_alloc()

Copy a metadata handle

Synopsis:

```
#include <mme/explore.h>

mme_metadata_hdl_t *mme_metadata_alloc(
                        const mme_metadata_hdl_t *metadata );
```

Arguments:

metadata A pointer to the metadata to copy.

Library:

mme

Description:

The function *mme_metadata_alloc()* makes and returns a malloced copy of a specified metadata handle structure **mme_metadata_hdl_t**, making it easier for users of the MME's explorer API to copy retrieved items.

The client application must deallocate the returned value from *mme_metadata_alloc()* by using *free()*.

For more information about managing metadata handles, see "Managing explorer structures and metadata handles" in the chapter Metadata and Artwork in the *MME Developer's Guide*.

Events

None delivered.

Blocking and validation

This function performs no validations and doesn't block.

Returns

Returns:

A copied metadata handle structure.

Success.

0 An error occurred (*errno* is set), or the metadata handle received is NULL.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_extract_data()*, *mme_metadata_extract_string()*, *mme_metadata_extract_unsigned()*, **`mme_metadata_hdl_t`**, *mme_ms_metadata_done()*, *mme_ms_metadata_get()*

mme_metadata_extract_data()

Get the data format metadata from the metadata handle

Synopsis:

```
#include <mme/metadata.h>

const void *mme_metadata_extract_data( const mme_metadata_hdl_t *metadata,
                                       const char *type,
                                       uint32_t flags,
                                       size_t *length );
```

Arguments:

metadata	The pointer to the handle with the metadata.
type	The type of metadata to extract. See METADATA_*.
flags	For future use.
length	A pointer to the location to which the function should return the length, in bytes, of the extracted data. If there is no data, this value is 0 (zero).

Library:

metadata

Description:

The function *mme_metadata_extract_data()* returns the format of the metadata retrieved by *mme_ms_metadata_get()* and placed in the metadata handle **mme_metadata_hdl_t**. Metadata formats are defined by the METADATA_FORMAT_* enumerated values.

Events

None delivered.

Blocking and validation

This function validates that the metadata handle isn't NULL. It doesn't block.

Returns:

Data in the character string, or NULL if no data is found (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_string()*, *mme_metadata_extract_unsigned()*, **mme_metadata_hdl_t**, *mme_ms_metadata_done()*, *mme_ms_metadata_get()*

mme_metadata_extract_string()

Get the string format from the metadata handle

Synopsis:

```
#include <mme/metadata.h>

const char *mme_metadata_extract_string( const mme_metadata_hdl_t *metadat
                                         const char *type,
                                         uint32_t flags );
```

Arguments:

metadata	The pointer to the handle with the metadata, returned by *mme_ms_metadata_get()*.
type	The type of metadata to extract. See METADATA_*.
flags	For future use.

Library:

metadata

Description:

The function *mme_metadata_extract_string()* extracts metadata in character string format from the metadata handle **mme_metadata_hdl_t**.

Events

None delivered.

Blocking and validation

This function validates that the metadata handle isn't NULL. It doesn't block.

Returns:

Data in the character string, or NULL if no data is found (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_data()*, *mme_metadata_extract_unsigned()*, **`mme_metadata_hdl_t`**, *mme_ms_metadata_done()*, *mme_ms_metadata_get()*

mme_metadata_extract_unsigned()

Get unsigned metadata from the metadata handle

Synopsis:

```
#include <mme/metadata.h>

int mme_metadata_extract_unsigned( const mme_metadata_hdl_t *metadata,
                                   const char *type,
                                   uint32_t flags,
                                   unsigned *value);
```

Arguments:

metadata	The pointer to the handle with the metadata, returned by *mme_ms_metadata_get()*.
type	The type of metadata to retrieve. See METADATA_*.
flags	For future use.
value	A pointer to the location where the value is to be returned; must *not* be NULL.

Library:

metadata

Description:

The function *mme_metadata_extract_unsigned()* extracts unsigned metadata from the metadata handle **mme_metadata_hdl_t**.

Events

None delivered.

Blocking and validation

This function validates that the metadata handle isn't NULL. It doesn't block.

Returns:

0	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_data()*, *mme_metadata_extract_string()*, **mme_metadata_hdl_t**, *mme_ms_metadata_done()*, *mme_ms_metadata_get()*

mme_metadata_create_session()

Create a new metatdata session

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_create_session( mme_hdl_t *hdl,
                                 mme_metadata_session_t **session );
```

Arguments:

hdl An MME connection handle.

session A pointer to the location with the metadata session structure.

Library:

mme

Description:

The function *mme_metadata_create_session()* creates a new metata session. Creating a metadata session guarantees that the images loaded and the metadata retrieved remain valid until the session is ended by a call to *mme_metadata_free_session()*.

A client application may have multiple metadata sessions open at the same time, only limited by system resources. Because every metadata session consumes system resources, the client application should end a metadata session when the data requested in that session is no longer needed.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

EOK and a valid pointer to an **mme_metadata_session_t** data structure.
Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_free_session(), *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **mme_metadata_image_url_t**, **mme_metadata_info_t**, **mme_metadata_session_t**

mme_metadata_free_session()

End a metadata session

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_free_session( mme_metadata_session_t *session );
```

Arguments:

session A pointer to a metadata session structure.

Library:

mme

Description:

The function *mme_metadata_free_session()* frees the memory and the images used in a metadata session.

Every metadata session consumes system resources. The client application should always call this function to end a metadata session when the data requested in that session is no longer needed.

Events

None delivered.

Blocking and validation

This function will cancel any pending metadata or image requests before returning. These cancellations may delay the return of the this function.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **`mme_metadata_image_url_t`**, **`mme_metadata_info_t`**, **`mme_metadata_session_t`**

mme_metadata_getinfo_current()

Get metadata for the currently playing track

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_getinfo_current( mme_metadata_session_t *session,
                                  const char *metadata_groups,
                                  uint64_t *mdinfo_rid,
                                  mme_metadata_info_t **metadata );
```

Arguments:

session	A pointer to a metadata session structure.
metadata_groups	A pointer to a string representing the metadata information groups for which metadata is requested.
mdinfo_rid	A pointer to a generated metadata information request ID.
metadata	A pointer to the location with the requested metadata. See "*metadata* pointer" below.

Library:

mme

Description:

The function *mme_metadata_getinfo_current()* retrieves metadata for the currently playing track and places it at the location specified by *metadata*. You must call *mme_metadata_create_session()* to create a metadata session before using *mme_metadata_getinfo_current()*.

There is no guarantee that the current track will not change between the time *mme_metadata_getcurrent()* is called and the return of the requested data. The client application must therefore monitor track change events, and make a new request for metadata if the track changes.

- Metadata and images retrieved with this function are only valid for the current metadata session.
- A call to an *mme_metadata_getinfo_*()* function switches the metadata session context to the newly requested file, thus causing any requests for image IDs from previous image data to fail.
- After an *mme_metadata_getinfo_*()* function has been called, any further calls to an *mme_metadata_getinfo_*()* function before receipt of a MME_EVENT_METADATA_INFO event will return an EBUSY error.

metadata pointer

The *metadata* argument points to a pointer to a **mme_metadata_info_t** metadata structure with the retrieved metadata. Depending on the value of *metadata*, *mme_metadata_getinfo_*()* operates either synchonously or asynchronously.

NULL pointer

If *metadata* is NULL, *mme_metadata_getinfo_*()* operates *a*synchronously, and the **mme_metadata_info_t** structure is delivered with the MME_EVENT_METADATA_INFO event.

non-NULL pointer

If *metadata* is non-NULL function *mme_metadata_getinfo_*()* operates synchronously and the following applies:

- If the referenced pointer to the **mme_metadata_info_t** structure is NULL, *mme_metadata_getinfo_*()* allocates memory for the structure.
- If the referenced pointer to the to the **mme_metadata_info_t**structure is non-NULL *mme_metadata_getinfo_*()* reuses the memory at the indicated locations, increasing the buffer for the structure as needed.

For an example of the XML delivered in the **mme_metadata_info_t** structure, see "XML content" with the description of the structure.

Events

MME_EVENT_METADATA_INFO.

Blocking and validation

See "*metadata* pointer" above.

Returns:

`0`	Success: *mdinfo_rid* is set.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **`mme_metadata_image_url_t`**, **`mme_metadata_info_t`**, **`mme_metadata_session_t`**

mme_metadata_getinfo_file()

Get metadata for a specified file, based on the filepath

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_getinfo_file( mme_metadata_session_t *session,
                               uint64_t msid,
                               const char *file,
                               const char *metadata_groups,
                               uint64_t *mdinfo_rid,
                               mme_metadata_info_t **metadata );
```

Arguments:

session	A pointer to a metadata session structure.
msid	The mediastore ID for the mediastore with the file for which metadata is required.
file	A pointer to the path, relative to the mediastore mountpath, of the file for which metadata is required.
metadata_groups	A pointer to a string representing the metadata information groups for which metadata is requested.
mdinfo_rid	A pointer to a generated metadata information request ID.
metadata	A pointer to the location with the requested metadata. See "*metadata* pointer" below.

Library:

mme

Description:

The function *mme_metadata_getinfo_file()* retrieves metadata for the file identified by the its filepath, and places this metadata at the location specified by *metadata*. You must call *mme_metadata_create_session()* to create a metadata session before using *mme_metadata_getinfo_file()*.

- Metadata and images retrieved with this function are only valid for the current metadata session.
- A call to an *mme_metadata_getinfo_*()* function switches the metadata session context to the newly requested file, thus causing any requests for image IDs from previous image data to fail.
- After an *mme_metadata_getinfo_*()* function has been called, any further calls to an *mme_metadata_getinfo_*()* function before receipt of a MME_EVENT_METADATA_INFO event will return an EBUSY error.

metadata pointer

The *metadata* argument points to a pointer to a **mme_metadata_info_t** metadata structure with the retrieved metadata. Depending on the value of *metadata*, *mme_metadata_getinfo_*()* operates either synchonously or asynchronously.

NULL pointer

If *metadata* is NULL, *mme_metadata_getinfo_*()* operates *a*synchronously, and the **mme_metadata_info_t** structure is delivered with the MME_EVENT_METADATA_INFO event.

non-NULL pointer

If *metadata* is non-NULL function *mme_metadata_getinfo_*()* operates synchronously and the following applies:

- If the referenced pointer to the **mme_metadata_info_t** structure is NULL, *mme_metadata_getinfo_*()* allocates memory for the structure.
- If the referenced pointer to the to the **mme_metadata_info_t**structure is non-NULL *mme_metadata_getinfo_*()* reuses the memory at the indicated locations, increasing the buffer for the structure as needed.

For an example of the XML delivered in the **mme_metadata_info_t** structure, see "XML content" with the description of the structure.

Events

MME_EVENT_METADATA_INFO.

Blocking and validation

See "*metadata* pointer" above.

Returns:

`0`	Success: *mdinfo_rid* is set.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **`mme_metadata_image_url_t`**, **`mme_metadata_info_t`**, **`mme_metadata_session_t`**

mme_metadata_getinfo_library()

Get metadata for a specified file, based on the file ID

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_getinfo_library( mme_metadata_session_t *session,
                                  uint64_t fid,
                                  const char *metadata_groups,
                                  uint64_t *mdinfo_rid,
                                  mme_metadata_info_t **metadata );
```

Arguments:

session	A pointer to a metadata session structure.
fid	The file ID of the file for which metadata is required.
metadata_groups	A pointer to a string representing the metadata information groups for which metadata is requested.
mdinfo_rid	A pointer to a generated metadata information request ID.
metadata	A pointer to the location with the requested metadata. See "*metadata* pointer" below.

Library:

mme

Description:

The function *mme_metadata_getinfo_library()* retrieves metadata for the file identified by the its file ID, and places this metadata at the location specified by *metadata*. You must call *mme_metadata_create_session()* to create a metadata session before using *mme_metadata_getinfo_library()*.

- Metadata and images retrieved with this function are only valid for the current metadata session.
- A call to an *mme_metadata_getinfo_*()* function switches the metadata session context to the newly requested file, thus causing any requests for image IDs from previous image data to fail.
- After an *mme_metadata_getinfo_*()* function has been called, any further calls to an *mme_metadata_getinfo_*()* function before receipt of a MME_EVENT_METADATA_INFO event will return an EBUSY error.

metadata pointer

The *metadata* argument points to a pointer to a **mme_metadata_info_t** metadata structure with the retrieved metadata. Depending on the value of *metadata*, *mme_metadata_getinfo_*()* operates either synchonously or asynchronously.

NULL pointer

If *metadata* is NULL, *mme_metadata_getinfo_*()* operates *a*synchronously, and the **mme_metadata_info_t** structure is delivered with the MME_EVENT_METADATA_INFO event.

non-NULL pointer

If *metadata* is non-NULL function *mme_metadata_getinfo_*()* operates synchronously and the following applies:

- If the referenced pointer to the **mme_metadata_info_t** structure is NULL, *mme_metadata_getinfo_*()* allocates memory for the structure.
- If the referenced pointer to the to the **mme_metadata_info_t**structure is non-NULL *mme_metadata_getinfo_*()* reuses the memory at the indicated locations, increasing the buffer for the structure as needed.

For an example of the XML delivered in the **mme_metadata_info_t** structure, see "XML content" with the description of the structure.

Events

MME_EVENT_METADATA_INFO.

Blocking and validation

See "*metadata* pointer" above.

Returns:

0 Success: *mdinfo_rid* is set.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **mme_metadata_image_url_t**, **mme_metadata_info_t**, **mme_metadata_session_t**

Synopsis:

```
#include <mme/metadata.h>

struct mme_metadata_hdl;
typedef struct mme_metadata_hdl mme_metadata_hdl_t;
```

Description:

The structure **mme_metadata_hdl_t** carries the metadata retrieved by *mme_metadata_extract_data()* and *mme_metadata_extract_string()*.

Creating and freeing the metadata handle

A metadata handle can be acquired through any of these functions:

- *mme_explore_info_get()*
- *mme_ms_metadata_get()*
- *mme_trksessionview_metadata_get()*

The data in the metadata handle can be used by *mme_metadata_extract_string()* and *mme_metadata_extract_data()*, and remains valid until the handle is freed.

To free a metadata handle, use one of these methods:

- Handles created by *mme_ms_metadata_get()* or *mme_trksessionview_metadata_get()*, call *mme_ms_metadata_done()*.
- Handles created by *mme_explore_info_get()*, call *mme_explore_end()*, or *mme_explore_info_get()* to create e new handle.

Classification:

QNX Multimedia

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_data()*, *mme_metadata_extract_string()*, *mme_metadata_extract_unsigned()*, *mme_ms_metadata_done()*, *mme_ms_metadata_get()*

mme_metadata_image_cache_clear()

Purge images from the image cache

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_image_cache_clear( mme_hdl_t *hdl,
                                    uint64_t msid );
```

Arguments:

hdl An MME connection handle.

msid The ID of the mediastore for which images must be purged from the image cache. Set to 0 (zero) to clear the entire cache.

Library:

mme

Description:

The function *mme_metadata_image_cache_clear()* clears from the image cache:

- all images associated with the specified mediastore; or,
- if *msid* is set to 0, all images in the cache

This function can be called at any time; you do *not* need to create a metadata session before clearing the image cache.

If a client application attempt to clear the cache while an item is being inserted into the cache, *mme_metadata_image_cache_clear()* returns an EBUSY error. If a client application receives this error, it should attempt to clear the cache again at a later time.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **`mme_metadata_image_url_t`**, **`mme_metadata_info_t`**, **`mme_metadata_session_t`**

mme_metadata_image_load()

Load an image for a file

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_image_load( mme_metadata_session_t *session,
                             uint64_t mdinfo_rid,
                             unsigned image_id,
                             int image_format_profile,
                             uint64_t *mdimage_rid,
                             mme_metadata_image_url_t **image_url );
```

Arguments:

session	A pointer to a metadata session structure.
mdinfo_rid	A metadata information request ID, obtained by a call to a *mme_metadata_getinfo_*()* function.
image_id	The ID of the image, obtained from the track metadata .
image_format_profile	Predefined profile format index. Set to -1 for no conversion.
mdimage_rid	A pointer to a generated metadata image request ID, populated on success.
image_url	A pointer to the location with the requested image. See "*image_url* pointer" below.

Library:

mme

Description:

The function *mme_metadata_image_load()* uses information retrieved by a call to any of the *mme_metadata_getinfo_*()* functions to load an image to the location specified by the URL referenced by *image_url*. You must call *mme_metadata_create_session()* to create a metadata session before using *mme_metadata_getinfo_current()*.

- Metadata and images retrieved with this function are only valid for the current metadata session.
- A call to an *mme_metadata_getinfo_*()* function switches the metadata session context to the newly requested file, thus causing any requests for image IDs from previous image data to fail.
- After an *mme_metadata_getinfo_*()* function has been called, any further calls to an *mme_metadata_getinfo_*()* function before receipt of a MME_EVENT_METADATA_INFO event will return an EBUSY error.

image_url pointer

The *image_url* argument points to a pointer to a **mme_metadata_image_url_t** metadata structure with the retrieved URL for the requested image. Depending on the value of *image_url*, *mme_metadata_image_load()* operates either synchonously or asynchronously.

NULL pointer

If *image_url* is NULL, *mme_metadata_getinfo_current()* operates *a*synchronously, and the **mme_metadata_info_t** structure is delivered with the MME_EVENT_METADATA_INFO event.

non-NULL pointer

If *image_url* is non-NULL function *mme_metadata_image_load()* operates synchronously and the following applies:

- If the referenced pointer to the **mme_metadata_info_t** structure is NULL, *mme_metadata_image_load()* allocates memory for the structure.
- If the referenced pointer to the to the **mme_metadata_info_t**structure is non-NULL *mme_metadata_image_load()* reuses the memory at the indicated locations, increasing the buffer for the structure as needed.

For an example of the XML delivered in the **mme_metadata_*_t** structure, see "XML content" with the description of the **mme_metadata_info_t** structure.

Events

MME_EVENT_METADATA_IMAGE.

Blocking and validation

See "*image_url* pointer" above.

Returns:

`0`	Success: *mdimage_rid* is set.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_unload()*, **`mme_metadata_image_url_t`**, **`mme_metadata_info_t`**, **`mme_metadata_session_t`**

mme_metadata_image_unload()

Clear image from temporary storage

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_image_unload( mme_metadata_session_t *session,
                               uint64_t mdimage_rid );,
```

Arguments:

session	A pointer to a metadata session structure.
mdimage_rid	A metadata image request ID, obtained by a call to a *mme_metadata_getinfo_*()* function.

Library:

mme

Description:

The function *mme_metadata_image_unload()* removes from temporary storage an image loaded by *mme_metadata_image_load()*. The image to remove from temporary storage is identified by the *mdimage_rid*, which was generated by a *mme_metadata_image_load()* function when it retrieved an image for a file.

If *mme_metadata_image_unload()* is called while an image is loading, the call cancels the load, and the MME delivers the event with the **mme_event_metadata_image_t** *error* member set to ECANCELED.

You must call *mme_metadata_create_session()* to create a metadata session before using *mme_metadata_unload()*.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, **mme_metadata_image_url_t**, **mme_metadata_info_t**, **mme_metadata_session_t**

mme_metadata_image_url_t

The structure carrying the URL for an image

Synopsis:

```
#include <mme/types.h>

typedef struct s_mme_metadata_image_url {
    int32_t   len;
    char      url[1];
} mme_metadata_image_url_t;
```

Description:

The structure **mme_metadata_image_url_t** carries the URL retrieved by *mme_metadata_image_load()* used with a synchronous connection. This URL can be used to load an image from a remote location.

Member	Type	Description
len	**int32_t**	The length, in bytes, of the *url* string, including its NULL terminator.
url	**char**	A NULL-terminated URL formated string location of an image.

Classification:

QNX Multimedia

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **mme_metadata_info_t**, **mme_metadata_session_t**

Synopsis:

```
#include <mme/types.h>

typedef struct s_mme_metadata_info {
    int  len;
    char xmlbuf[1];
} mme_metadata_info_t;
```

Description:

The structure **mme_metadata_info_t** carries the metadata retrieved by *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()* and *mme_metadata_getinfo_library()*.

Member	Type	Description
len	**int**	The length, in bytes, of the *xmlbuf* string, including its NULL terminator. See "XML content" below.
xmlbuf	**char**	A NULL-terminated XML formated string containing metadata.

XML content

The MME's metadata API organizes metadata into groups and subgroups. You can use these groups and subgroups to request only the metadata you need, thereby optimizing performance and reducing resource consumption.

To request only specified metadata, use the following guidelines to set the character string referenced by a *mme_metadata_getinfo_*()* function's *metadata_groups* argument:

- Setting the *metadata_groups* argument to NULL, or the group to **"*"** instructs the function to return *all* avaialble metadata for the file.
- A metadata group can use wildcards characters to obtain all metadata for a subgroup. For example, to get all image subgroups, use the string **"image/*"**.

Supported <format> attributes

The table below list the attributes for the **<format>** element currently supported by the MME's metadata API.

Attribute	Optional	Description
height	Yes	The image height, in pixels.

continued...

Attribute	Optional	Description
width	Yes	The image width, in pixels.
mime_type	Yes	The content MIME type.
start_timepos	Yes	The image start time, in milliseconds, from the start of the track.
end_timepos	Yes	The image end time, in milliseconds, from the start of the track.
desc	Yes	An image description.
size	Yes	The image size, in bytes.
url	Yes	An external URL to the image.

Example: default XML content

Below a example of the default XML content returned in *xmlbuf* by a call to an *mme_metadata_getinfo_*()* function. No metadata group is enabled:

```
<?xml version="1.0" standalone="yes"?>
<container type="file">
    <track index="0">
        <audio>
            <stream index="0"/>
        </audio>
        <images>
            <image index="0"/>
            <image index="1"/>
        </images>
    </track>
</container>
```

Example: XML content with one metadata group enable

Below is an example of the XML content returned in *xmlbuf* by a call to an *mme_metadata_getinfo_*()* function. Only the **<image>**/**<format>** metadata group is enabled:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<container type="file">
    <track index="0">
        <audio>
            <stream index="0"/>
        </audio>
        <images>
            <image index="0">
                <format width="0" height="0" size="29316"/>
            </image>
            <image index="1"/>
        </images>
    </track>
</container>
```

Classification:

QNX Multimedia

See also:

mme_metadata_create_session(), mme_metadata_free_session(), mme_metadata_getinfo_current(), mme_metadata_getinfo_file(), mme_metadata_getinfo_library(), mme_metadata_image_cache_clear(), mme_metadata_image_load(), mme_metadata_image_unload(), **mme_metadata_image_url_t**, **mme_metadata_session_t**

mme_metadata_session_t

A metadata session identifier

Synopsis:

```
#include <mme/types.h>

typedef struct s_mme_metadata_session {
    uint64_t session_id;
} mme_metadata_session_t;
```

Description:

The structure **mme_metadata_session_t** carries a unique identifier withinforamtion about a metadata session. It is set by *mme_metadata_create_session()* and used by the *mme_metadata_*()* functions. It is cleared by *mme_metadata_free_session()*.

Member	Type	Description
session_id	**uint64_t**	A metadata session identifier.

Classification:

QNX Multimedia

See also:

mme_metadata_create_session(), *mme_metadata_free_session()*, *mme_metadata_getinfo_current()*, *mme_metadata_getinfo_file()*, *mme_metadata_getinfo_library()*, *mme_metadata_image_cache_clear()*, *mme_metadata_image_load()*, *mme_metadata_image_unload()*, **mme_metadata_image_url_t**, **mme_metadata_info_t**

mme_metadata_set()

Set the metadata for a file

Synopsis:

```
#include <mme/mme.h>

int mme_metadata_set( mme_hdl_t *hdl,
                      uint64_t fid,
                      mm_metadata_t *metadata,
                      uint64_t flags );
```

Arguments:

hdl	An MME connection handle.
fid	The file ID of the file whose metadata you want to set.
metadata	A pointer to the structure that carries the file metadata. For more information, see **mm_metadata_t**.
flags	A flag to define the behavior of the call. For future use.

Library:

mme

Description:

The function *mme_metadata_set()* sets the metadata in the MME database for a specified file. The client application can use this function with an HMI to allow the end-user to change the metadata in the MME database for copied and ripped media. It sets the metadata in the database, and can be used to correct and complete metadata that was incorrectly or incompletely entered when the file was copied or ripped.

To set the metadata for a file:

1. Complete the structure **mm_metadata_t** with the file metadata.
2. Call *mme_metadata_set()*, specifying the file ID.

Events

None delivered.

Blocking and validation

This function performs no validations, and doesn't block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_mediacopier_add(), *mme_mediacopier_get_mode()*, *mme_mediacopier_set_mode()*,

Synopsis:

```
#include <mme/types.h>

typedef enum mme_mode_random {
    ...
} mme_mode_random_t;
```

Description:

The enumerated type **mme_mode_random_t** defines random mode settings. These settings match the settings used by iPods:

- MME_RANDOM_OFF (0) — random mode is not selected
- MME_RANDOM_ALL (1) — random playback for the track session
- MME_RANDOM_ALBUMS (2) — random playback for the current album or directory on an iPod device. The MME doesn't support this mode, and falls back to MME_RANDOM_ALL if this mode is set. However, if playback is handled externally (i.e. by an iPod device), then the random command is handled by the device.
- MME_RANDOM_FOLDER (3) — random playback for the current folder
- MME_RANDOM_SUBFOLDER (4) — random playback for the current subfolder

For more information about playback random mode, see *mme_setrandom()* and "Using random and repeat modes" in the chapter Playing Media of the *MME Developer's Guide*.

Classification:

QNX Multimedia

See also:

mme_mode_repeat_t, *mme_getrandom()*, *mme_getrepeat()*, *mme_getscanmode()* *mme_setrandom()* *mme_setrepeat()*

mme_mode_repeat_t

Repeast mode values

Synopsis:

```
#include <mme/types.h>

typedef enum mme_mode_repeat_t;
```

Description:

The enumerated type **mme_mode_repeat_t** defines random mode settings. These settings match the settings used by iPods:

- MME_REPEAT_OFF — repeat mode is not selected
- MME_REPEAT_SINGLE — repeat the current track
- MME_REPEAT_ALL — repeat all tracks in the track session
- MME_REPEAT_FOLDER — repeat all tracks in the current folder
- MME_REPEAT_SUBFOLDER — repeat all tracks in the current subfolder

For more information about playback repeat mode, see *mme_setrepeat()* and "Using random and repeat modes" in the chapter Playing Media of the *MME Developer's Guide*.

Classification:

QNX Multimedia

See also:

mmme_mode_random_t, *mme_getrandom()*, *mme_getrepeat()*, *mme_getscanmode()* *mme_setrandom()*, *mme_setrepeat()*

Synopsis:

```
#include <mme/interface.h>

#define _MME_MSCAP_*_MASK
#define MME_MSCAP_*
```

Description:

The constants MME_MSCAP_* are bit masks defining mediastore capabilities. The values listed in the table below are used by the *capabilities* field in the **mediastores** table.

Constant	Value	Description
MME_MSCAP_SYNC	0x00000001	The mediastore can be synchronized.
MME_MSCAP_PRUNABLE	0x00000002	Synchronization should manage pruning of this mediastore.
MME_MSCAP_SYNC_DIRECTED	0x00000004	The mediastore supports directed synchronizations.
MME_MSCAP_NO_AUTO_SYNC	0x00000008	The mediastore is never automatically synchronized.
MME_MSCAP_PRIO_FOLDER	0x00000010	The mediastore can prioritize folders for synchronization.
MME_MSCAP_MEDIAFS_1WIRE	0x00000020	The device is a media device.
MME_MSCAP_MEDIAFS_2WIRE	0x00000040	The device is a media device.
MME_MSCAP_DEVICE_TRACKSESSIONS	0x00000080	The device manages its own track sessions.
MME_MSCAP_NOWPLAYING_METADATA	0x00000100	Metadata for the currently playing track can be retrieved from the device.
MME_MSCAP_NOWPLAYING_FILENAME	0x00000200	The filename for the currently playing track can be retrieved from the device.
MME_MSCAP_DEVICE_SAVES_STATE	0x00000400	The device can save its own state; used for resuming playback with *mme_play_resume_msid()*.

continued...

Constant	Value	Description
MME_MSCAP_DEVICE_REPEATRANDOM	0x00000800	The device supports repeat and random modes. This capability does not apply to USB devices; it applies *only* to devices with the MME_MSCAP_DEVICE_TRACKSESSIONS capability set.
MME_MSCAP_DELETE_ON_EJECT	0x00001000	The MME should delete entries for this mediastore when it is ejected.
MME_MSCAP_PLAY_FILE	0x00002000	The device supports the deprecated *mme_play_file()* function.
MME_MSCAP_EXPLORABLE	0x00004000	The device supports the MME's explorer API. See *mme_explore_start()* and the other *mme_explore_*()* functions.
MME_MSCAP_TRKSESSIONVIEW_METADATA	0x00008000	The device supports the *mme_trksessionview_metadata_get()* function.
MME_MSCAP_TRACK_POSITION_COOKIE_BASED	0x00010000	The device supports the See *mme_trksession_save_state()* function.
MME_MSCAP_SUPPORTS_VIDEO	0x00020000	The device supports video playback.
MME_MSCAP_CONNECTION_NONOPTIMAL	0x00040000	The device is not using the optimal link; for example, an iPod that supports USB is using a serial transport.
MME_MSCAP_AUDIO_NONOPTIMAL	0x00080000	The device is not using the optimal audio link; for example, an iPod that supports digital audio is using analog audio.
MME_MSCAP_SET	0x80000000	Device capabilities have been set (make non-zero).

For more information about detecting mediastores and discovering their capabilities, see "Mediastore and device capabilities" in the chapter Working with Mediastores of the *MME Developer's Guide*.

Classification:

QNX Multimedia

See also:

MME_FORMAT_*, MME_FTYPE_*, MME_STORAGETYPE_*, MME_SYNC_OPTION_*

mme_ms_clear_accurate()

Mark library entries as inaccurate

Synopsis:

```
#include <mme/mme.h>

int mme_ms_clear_accurate( mme_hdl_t *hdl,
                           uint64_t msid);
```

Arguments:

hdl An MME connection handle.

msid The ID for the mediastore to be marked inaccurate.

Library:

mme

Description:

The function *mme_ms_clear_accurate()* clears the *accurate* fields in the **library** for items linked to the specified mediastore. Clearing the **accurate** marks the entry in the **library table** as inaccurate, so that the MME synchronizers will update the data.

Set *msid* to 0 to mark as inaccurate all entries in the **library** linked to all mediastores.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), mme_resync_mediastore(), mme_setpriorityfolder(), mme_sync_cancel(), mme_sync_directed(), mme_sync_file(), mme_sync_get_msid_status(), mme_sync_get_status()

mme_ms_metadata_done()

Clear the metadata handle

Synopsis:

```
#include <mme/metadata.h>

void mme_ms_metadata_done( mme_metadata_hdl_t *metadata );
```

Arguments:

metadata The pointer to the handle with the metadata.

Library:

metadata

Description:

The function *mme_ms_metadata_done()* clears the metadata handle. It should be used when the metadata in the handle is no longer needed.

Events

None delivered.

Blocking and validation

This function validates that the metadata handle isn't NULL. It doesn't block.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_data()*, *mme_metadata_extract_string()*, **mme_metadata_hdl_t**, *mme_ms_metadata_get()*

Synopsis:

```
#include <mme/metadata.h>

mme_metadata_hdl_t *mme_ms_metadata_get(
                        mme_hdl_t *hdl,
                        uint64_t *msid,
                        const char *path,
                        const char *types,
                        uint32_t flags );
```

Arguments:

hdl	The MME connection handle.
msid	The ID of the mediastore with the file whose metadata is required.
path	The path and filename (not including the mediastore mountpath) of the file whose metadata is required.
types	A pointer to a string containing a comma-separated list of metadata types to retrieve. May *not* be NULL. See METADATA_*.
flags	For future use.

Library:

metadata

Description:

The function *mme_metadata_get()* gets metadata for a file and places it in the metadata handle **mme_metadata_hdl_t**. The type of metadata retrieved is defined by the METADATA_FORMAT_* enumerated values.

Events

None delivered.

Blocking and validation

This function performs no validations and doesn't block.

Returns:

Data in the character string, or NULL if no data is found (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

METADATA_*, *mme_metadata_alloc()*, *mme_metadata_extract_data()*, *mme_metadata_extract_string()*, **`mme_metadata_hdl_t`**, *mme_ms_metadata_done()*

Synopsis:

```
#include <mme/mme.h>

int mme_ms_restart( mme_hdl_t *hdl,
                    uint64_t msid );
```

Arguments:

db An MME connection handle.

msid The ID of the mediastore to restart.

Library:

mme

Description:

The function *mme_ms_restart()* causes the specified mediastore to go through an "active" to "nonexistent" transition, followed by an insertion to the "active" state.

When the state of a mediastore changes from another state to "nonexistent", the MME always prunes the entries for that mediastore from its database, *no matter what the pruning configurations*. Thus, when *mme_ms_restart()* is successful, when the mediastore restarts it appears to the MME as a *new* mediastore, and the MME assigns it a new mediastore ID.

CAUTION: *mme_ms_restart()* is:

- *not* the recommended method for rediscovering a mediastore. It may be changed or removed from the MME API.
- not supported for mediastores that are not active, or for mediastores that use an **mmdev** handler plugin.

Events

None delivered.

Blocking and validation

This function validates the request and runs asynchronously, so it may fail after returning success. The calling application must examine the mediastore state change events to determine if the entire operation finished successfully.

Calls using that MME handle used by *mme_ms_restart()* will fail until the operation is complete, even if the call to *mme_ms_restart()* has returned.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set):

- EINVAL — the mediastore does not exist or is not active
- ENOTSUP — the mediastore uses an **mmdev** handler

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

"Mediastore states" in the chapter Working with Mediastores of the *MME Developer's Guide*

Synopsis:

```
#include <mme/types.h>

typedef enum {
e_mme_ms_nonexistent = 0,
e_mme_ms_unavailable,
e_mme_ms_available,
e_mme_ms_active
} mme_ms_state_t;
```

Description:

The enumerated type **mme_ms_state_t** defines mediastore states:

- e_mme_ms_nonexistent — non-existent: the MME has no database entry for the mediastore.
- e_mme_ms_unavailable — unavailable: the MME has a database entry for the mediastore, but the mediastore is not in the system in which the MME is running.
- e_mme_ms_available — available: the MME has a database entry for the mediastore, and the mediastore is in the system in which the MME is running. That is, the MME knows the location of the mediastore, but the mediastore cannot be synchronized, and tracks on the mediastore cannot be ripped or played. This state is generally possible only for disk-based media stores in multi-disk changers.
- e_mme_ms_active — active: the usable state of a mediastore. The MME has a database entry for the mediastore, the mediastore can be synchronized, and tracks on the mediastore can be ripped or played

For more information about mediastore states and state transitions, see the chapter Working with mediastores of the *MME Developer's Guide*.

Classification:

QNX Multimedia

See also:

mme_ms_statechange_t

mme_ms_statechange_t

Data for media store state change event

Synopsis:

```
#include <mme/types.h>

typedef struct s_mme_ms_statechange {
    uint64_t    msid;
    uint32_t    old_state;
    uint32_t    new_state;
    uint16_t    device_type;
    uint16_t    storage_type;
    uint32_t    reserved;

}
```

Description:

The structure **mme_ms_statechange_t** carries data for the mediastore state change events MME_MS_STATECHANGE. It includes at least the members described in the table below.

Member	Type	Description
msid	**uint64_t**	The mediastore ID
old_state	**uint32_t**	The previous state of the mediastore
new_state	**uint32_t**	The new state of the mediastore
device_type	**uint16_t**	The device type. See “Device types” below.
storage_type	**uint16_t**	The mediastore storage type, as defined by the MME_STORAGETYPE_* constant.
reserved	**uint32_t**	Reserved for internal use.

Mediastore states are defined by the enumerated value **mme_ms_state_t**. For more information about mediastore states and state transitions, see the chapter Working with Mediastores.

Device types

The value of *device_type* is defined by the *slottype* field for the mediastore in the **slots** table. This field uses the values defined by the MME_SLOTTYPE_*, and its use is defined by the user.

If the MME is unable to associate a mediastore that is available but not active with an entry in the **slots** table, the value for *device_type* may be MME_SLOTTYPE_UNKNOWN.

Classification:

QNX Multimedia

See also:

mme_ms_state_t

mme_newtrksession()

Create a new track session

Synopsis:

```
#include <mme/mme.h>

int mme_newtrksession( mme_hdl_t *hdl,
                       char *statement,
                       short int mode,
                       uint64_t *trksessionid );
```

Arguments:

hdl	An MME connection handle.
statement	An SQL statement that defines the track session you want to create.
mode	The track session mode. This mode can be either MME_PLAYMODE_LIBRARY (0) or MME_PLAYMODE_FILE (1).
trksessionid	The pointer to the location where the function can store the new track session ID. Pass this value to *mme_settrksession()* to activate the track session.

Library:

mme

Description:

The function *mme_newtrksession()* creates a new track session for the specified control context.

The SQL query passed to this function can select tracks from the **library** table, the **playlist** table, or any other valid source. The MME adds each new track session created by *mme_newtrksession()* to the **trksessions** table in the MME library.

The SQL statement should not end with a semicolon. The statement is actually a sub-statement, which *mme_newtrksession()* places into a larger statement. The result for the statement you pass to *mme_newtrksession() must* include a *fid* column.

For best performance, compose the query to look for media files only on available mediastores. For example, for library-mode track sessions, compose the query:

```
SELECT fid FROM library WHERE msid IN
    (SELECT msid FROM mediastores WHERE available=1)
```

For file-based track sessions, compose a query that returns the FTYPE_DEVICE *fid* for the mediastore with the files discovered through the explorer API. For example:

```
SELECT fid FROM library WHERE ftype=5 AND msid=3
```

For more information about library-mode and file-based track sessions, see "Working with track sessions" in the *MME Developer's Guide*.

After you have created a new track session, you need to:

- call *mme_settrksession()* to make it the active track session on the specified zone
- call *mme_play()* to start playing tracks in the track session

A new track session inherits its random and repeat modes from the control context in which it is created. For more information about these modes, see *mme_setrandom()* and *mme_setrepeat()*.

You can call *mme_trksession_get_info()* to get the ID of the active track session in a specific control context.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Examples:

```
// Create a new track session of all songs from a playlist
//  that are currently available
sql = qdb_mprintf(
        "SELECT fid FROM playlistdata WHERE "
        "plid = (SELECT plid FROM playlists WHERE name = '%q') "
        "AND msid IN (SELECT msid FROM mediastores WHERE available=1)",
        playlistname);

mme_newtrksession( &mme, sql, MME_PLAYMODE_LIBRARY, &trksessionid );
rc = mme_newtrksession( &mme, sql, MME_PLAYMODE_LIBRARY, &trksessionid );
if (rc == -1) {
    fprintf(stderr, "error creating new track session;");
    exit(1);
}

mme_settrksession( &mme, trksessionid );

// pass in a fid of 0 to start from the beginning.
mme_play(&mme, 0);
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_get_info(), mme_rmtrksession(), mme_settrksession()

Synopsis:

```
#include <mme/mme.h>

int mme_next( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_next()* skips to the next track in the currently playing track session.

Effect of play modes on behavior

The behavior of *mme_next()* is affected by the play modes set for the specified control context (sequential versus random, and repeat versus no repeat).

If sequential mode is set, the next track in the track session is determined by the *sequentialid* field in the next row of the **trksessionview** table. The order of the file IDs in this table column is determined by the **ORDER BY** clause used to create the track session.

If random mode is set, the next track in the track session is determined by the *randomid* field in the next row of the **trksessionview** table. The order of the file IDs in this table column is generated by the MME when it sets the track session.

Effect of repeat mode on the last track of a session

When the last track in the track session is playing, the result of calling *mme_next()* depends on whether the repeat mode is set.

If repeat is off, *mme_next()* sets *errno* to ENODATA when it has reached the end of the track session (or, when random mode is set, when all songs in the trackssessin have been played).

If repeat is on:

- if sequential mode is set, the MME plays the first track in the track session, as determined by the *sequentialid* column in the **trksessionview** table
- if random mode is set, the MME plays the first track in the track session, as determined by the *randomid* column in the **trksessionview** table

Working with an iPod device

iPod devices manage their own track sessions. To move to the next or previous track in an iPod track session, call the *mme_button()* function with **mm_button_t** set to MM_BUTTON_NEXT or MM_BUTTON_PREV, as required.

Events

Any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

Verifies that the *fid* is valid. Does not verify that the file exists, or that it is playable.

This function blocks on control contexts. If *mme_next()* is called and another function is called before *mme_next()* returns, the second function blocks on **io-media** until *mme_next()* returns. If there are no other pending calls, *mme_next()* returns without blocking on **io-media**.

Returns:

≥0 Success: *errno* set to ENODATA indicates that there are no more tracks to play.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_prev(), *mme_setrandom()*, *mme_setrepeat()*

mme_output_attr_t

Media output attributes

Synopsis:

```
#include <mme/types.h>

typedef struct mme_output_attr {
    union {
        struct {
            int       volume;
            int       balance;
            int       fade;
            int       mute;
            uint64_t  delay;
        } audio;

        struct {
            int       layer;
        } video;

        struct {
            /* not yet implemented */
        } encoded;
    };
} mme_output_attr_t;
```

Description:

The structure **mme_output_attr_t** carries playback output attributes and is used for getting and setting attributes on output devices. It is a union of the structures **audio**, **video** and **encoded**, and can therefore only control one class of output device at a time.

The members of the structures **audio**, **video** and **encoded** that make up **mme_output_attr_t** are described in the table below.

Structure	Member	Type	Description
mme_output_attr_t	*audio*	**struct**	Audio information
mme_output_attr_t	*video*	**struct**	Video information
mme_output_attr_t	*encoded*	**struct**	Encoding information. For future use.
audio	*volume*	**int**	The output volume, as a percent from 0 to 100.
audio	*balance*	**int**	The output balance: 0 (left); 50 (center); 100 (right).

continued...

Structure	Member	Type	Description
audio	*fade*	**int**	The output fade setting: 0 (back); 50 (center); 100 (forward).
audio	*mute*	**int**	The output muted setting: Set to **1** for muted, **0** for not muted.
audio	*delay*	**uint64_t**	The output delay, in millisenconds.
video	*layer*	**int**	The GF/video layer.

Classification:

QNX Multimedia

See also:

mme_play_get_output_attr()

mme_output_set_permanent()

Set the permanency status of an output device

Synopsis:

```
#include <mme/mme.h>

int mme_output_set_permanent( mme_hdl_t *hdl,
                              uint64_t outputid,
                              int permanent);
```

Arguments:

hdl	An MME connection handle.
outputid	The ID of the output device whose permanency status is to be set.
permanent	The output device's permanency status: Set this argument to 1 for permanent, 0 for not permanent.

Library:

mme

Description:

The function *mme_output_set_permanent()* sets the permanency status of the specified output device:

1 The output device is permanent.

0 The output device is *not* permanent.

Events

None delivered.

Blocking and validation

This function is fully validating and runs to completion.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_attach_output(), *mme_play_detach_output()*, *mme_play_get_zone()*, *mme_play_set_zone()*, *mme_zone_create()*, *mme_zone_delete()*

mme_outputtype_t

Define media output types

Synopsis:

```
#include <mme/types.h>

typedef enum mme_outputtype {
   MME_OUTPUTTYPE_UNKNOWN = 0,
   MME_OUTPUTTYPE_AUDIO = 1,
   MME_OUTPUTTYPE_VIDEO = 2,
   MME_OUTPUTTYPE_ENCODED = 3
} mme_outputtype_t;
```

Description:

The enumerated types **mme_outputtype_t** defines media output types. Its values are listed below:

- MME_OUTPUTTYPE_UNKNOWN
- MME_OUTPUTTYPE_AUDIO
- MME_OUTPUTTYPE_VIDEO
- MME_OUTPUTTYPE_ENCODED

Classification:

QNX Multimedia

See also:

mme_output_attr_t, *play_get_output_attr()*

Synopsis:

```
#include <mme/mme.h>

int mme_play( mme_hdl_t *hdl,
              uint64_t fid );
```

Arguments:

hdl An MME connection handle.

fid The file ID of the file or track you want to play. Pass as 0 to start playback at the first track in the track session.

Library:

mme

Description:

The function *mme_play()* plays tracks in a track session. This function can only be used after the client application has called *mme_newtrksession()* to create a track session, and *mme_settrksession()* to set the track session.

If you specify the *fid* in a library-based track-session, the MME starts playback with the specified *fid*. If the library-based track session contains more than one instance of the specified *fid*, the MME starts playback at the first instance of this *fid*.

The MME control context notifies the client application at set intervals while it is playing a track session by delivering the event MME_EVENT_TIME. You can change this period through the function *mme_set_notification_interval()*.

- If you need the file ID (*fid*) of the track being played, your client application can do one of the following:
  - wait for the MME_EVENT_TRACKCHANGE event, delivered when the track session starts playing a new track. This event contains the *fid*
  - call the function *mme_play_get_info()* and get the *fid* from **`mme_play_info_t.fid`**
- If you call *mme_play()* while a track is playing, the MME will drop the current track and start playing the new track.
- If *mme_play()* is unable to play a track in a track session it generates an MME_PLAY_ERROR_* event, then attempts to play the next track in the track session.
- If you attempt to play a file ID (*fid*) that is not in your track session, the MME will play the first track in the track session. This behavior is specific to MME 1.1.0; in subsequent releases, *mme_play()* will return an error.

Events

This function may deliver any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

This function does not verify that the *fid* is in the track session. If the connection to the MME is synchronous, the function validates that the file exists and that it is playable.

This function blocks on control contexts. If *mme_play()* is called and another function is called before *mme_play()* returns, the second function blocks on **`io-media`** until *mme_play()* returns. If there are no other pending calls, *mme_play()* returns without blocking on **`io-media`**.

Returns:

`≥0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_newtrksession(), *mme_next()*, *mme_prev()*, *mme_stop()*

mme_play_attach_output()

Attach an output to a zone

Synopsis:

```
#include <mme/mme.h>

int mme_play_attach_output( mme_hdl_t *hdl,
                            uint64_t zoneid,
                            uint64_t outputid );
```

Arguments:

hdl	An MME connection handle.
zoneid	The zone to which you want to attach the output device. If set to 0, use the current control context zone.
outputid	The ID of the output device to attach to the zone.

Library:

mme

Description:

The function *mme_play_attach_output()* attaches an output device to a specified zone. Playback on the control context using the specified zone will go to the output devices attached to that zone.

The MME saves the output device setting so that the next time the control context is used it will automatically send its output to the same output devices.

Events

None delivered.

Blocking and validation

This function blocks on control contexts. It validates parameters. In asynchronous mode, it returns before calling **io-media**.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_output_set_permanent(), *mme_play_detach_output()*, *mme_play_get_zone()*, *mme_play_set_zone()*, *mme_zone_create()*, *mme_zone_delete()*.

mme_play_bookmark()

Start playback from a bookmark

Synopsis:

```
#include <mme/mme.h>

int mme_bookmark_play( mme_hdl_t *hdl,
                       uint64_t bookmarkid );
```

Arguments:

hdl	An MME connection handle.
bookmarkid	The bookmark ID from which to play.

Library:

mme

Description:

The function *mme_play_bookmark()* begins playing a track from the specified bookmark. Its behavior is like that of *mme_play()*, except that instead of playing the track from its beginning, *mme_play_bookmark()* starts playback from the bookmark.

Like *mme_play()*, *mme_play_bookmark()* requires that the track to be in the current track session. In addition, the track must have the specified bookmark.

Events

This function may deliver any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

This function verifies that the *fid* is valid. It doesn't verify that the file exists, or that it is playable.

This function blocks on control contexts. If *mme_play()* is called and another function is called before *mme_play()* returns, the second function blocks on **io-media** until *mme_play()* returns. If there are no other pending calls, *mme_play()* returns without blocking on **io-media**.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_bookmark_create(), mme_bookmark_delete(),

mme_play_detach_output()

Detach an output from a zone

Synopsis:

```
#include <mme/mme.h>

int mme_play_detach_output( mme_hdl_t *hdl,
                            uint64_t zoneid,
                            uint64_t outputid );
```

Arguments:

hdl	An MME connection handle.
zoneid	The zone from which you want to detach the output device. If set to 0, use the current control context zone.
outputid	The ID of the output device to detach from the zone. .

Library:

mme

Description:

The function *mme_play_detach_output()* detaches an output device from a specified zone.

Events

None delivered.

Blocking and validation

This function blocks on control contexts. It validates parameters. In asynchronous mode, it returns before calling **io-media**.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_output_set_permanent(), *mme_play_attach_output()*, *mme_play_get_zone()*, *mme_play_set_zone()*, *mme_zone_create()*, *mme_zone_delete()*

This function is deprecated. Use file-based track sessions; see "Creating and modifying file-based track sessions" in the *MME Developer's Guide*.

Synopsis:

```
#include <mme/mme.h>

int mme_play_file( mme_hdl_t *hdl,
                   uint64_t msid,
                   const char *filename );
```

Arguments:

hdl	The handle of the control context.
msid	The ID of the mediastore with the track to be played.
filename	The path and filename of the track to play. The filename includes the path to the file on the mediastore, but it does *not* include the mountpath to the mediastore. The path in *filename* must begin with a "/" (slash). For example: **/songs_folder/album_folder/**.

Library:

mme

Description:

The function *mme_play_file()* plays a track on a mediastore regardless of whether the mediastore has been synchronized. This function can only be used to play a track on a mediastore that has its *capabilities* field in the **mediastores** table set to MME_MSCAP_PLAY_FILE.

Like *mme_play()*, in order to play a track, *mme_play_file()* requires a track session to be set, but does not require the track to be in the set track session.

Events

This function may deliver any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

This function does not verify that the *fid* is in the track session. If the connection to the MME is synchronous, the function validates that the file exists and that it is playable.

This function blocks on control contexts. If *mme_play_file()* is called and another function is called before *mme_play_file()* returns, the second function blocks on **io-media** until *mme_play_file()* returns. If there are no other pending calls, *mme_play_file()* returns without blocking on **io-media**.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play()

mme_play_get_info()

Get information about the track or file currently being played

Synopsis:

```
#include <mme/mme.h>

int mme_play_get_info( mme_hdl_t *hdl,
                       mme_play_info_t *info );
```

Arguments:

hdl An MME connection handle.

info A pointer to a **mme_play_info_t** structure that *mme_play_get_info()* can fill with the playback information.

Library:

mme

Description:

The function *mme_play_get_info()* retrieves current information about the track that is currently being played, and fills out the structure pointed to by *info*. For information about this structure, see **mme_play_info_t** in this reference.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥0 Success: MME retrieved the information for the current track and placed this information in the structure **mme_play_info_t**.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), *mme_play_get_status()*, *mme_play_get_info()*, *mme_play_get_status()*, *mme_play_set_speed()*, *mme_set_notification_interval()*

mme_play_get_output_attr()

Get the attributes for an output device

Synopsis:

```
#include <mme/mme.h>

int mme_play_get_output_attr( mme_hdl_t *hdl,
                              uint64_t outputdeviceid,
                              mme_output_attr_t *attr );
```

Arguments:

hdl	An MME connection handle.
outputdeviceid	The ID of the output device for which to get attibutes.
attr	A pointer to a structure with the output device attributes.

Library:

mme

Description:

The function *mme_play_get_output_attr()* gets the output attributes for the specified output device, and places them in a structure **mme_output_attr_t**. For more information about this structure, see **mme_output_attr_t** in this reference.

Events

None delivered.

Blocking and validation

This function blocks on control contexts.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_set_output_attr(), mme_output_set_permanent()

mme_play_get_speed()

Gets playback speed and direction (forward, reverse, pause) for tracks

Synopsis:

```
#include <mme/mme.h>

int mme_play_get_speed( mme_hdl_t *hdl,
                        int *speed );
```

Arguments:

hdl An MME connection handle.

speed A pointer to the playback speed for the current track, expressed in units of 1/1000 of normal speed.

Library:

mme

Description:

The function *mme_play_get_speed()* gets the playback speed for the current track or file.

The playback speed is expressed in units of 1/1000 of normal speed: 1000 means normal speed, 2000 means double speed, etc. Positive values mean forward, negative values mean reverse, and zero means pause. Values between 0 and 1000 are slow speed playback.

iPods do not report their current playback speed. Queries for their playback speed always return a nominal 1000, but this value should not be considered accurate.

Events

None delivered.

Blocking and validation

This function validates all data, and doesn't block.

Returns:

≥**0** Success: the playback speed was set.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), mme_play_get_info(), mme_play_get_status(), mme_play_set_speed()

mme_play_get_status()

Get the status of the current track

Synopsis:

```
#include <mme/mme.h>

int mme_play_get_status( mme_hdl_t *hdl,
                         mme_play_status_t *play_status);
```

Arguments:

hdl	An MME connection handle.
play_status	The pointer to the structure with the playback status information filled in by *mme_play_get_status()*.

Library:

mme

Description:

The function *mme_play_get_status()* retrieves the status of a media play. It provides the total play time of the media track and the play time elapsed by filling in the structure **mme_play_status_t** pointed to by *play_status*. See **mme_play_status_t** in this reference.

Events

None delivered.

Blocking and validation

This function validates all data, and doesn't block.

Returns:

≥**0**	Success: MME retrieved the status of the media play and filled in the information in the structure **mme_play_status_t**.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), *mme_play_get_info()*, *mme_play_get_status()*, *mme_play_set_output_attr()*, **`mme_time_t`**, **`mme_playstate_t`**, **`mme_playstatus_t`**

mme_play_get_zone()

Get the zone ID used by a control context

Synopsis:

```
#include <mme/mme.h>

int mme_play_get_zone( mme_hdl_t *hdl,
                       uint64_t *zoneid );
```

Arguments:

hdl	An MME connection handle.
zoneid	The zone ID.

Library:

mme

Description:

The function *mme_play_get_zone()* gets the zone used by the current control context. For more information about zones, see *mme_zone_create()*.

Events

None delivered.

Blocking and validation

This function is fully validating and runs to completion.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_attach_output(), *mme_play_detach_output()*, *mme_output_set_permanent()*, *mme_play_set_zone()*, *mme_zone_create()*, *mme_zone_delete()*

mme_play_info_t

Information about the currently playing track

Synopsis:

```
#include <mme/types.h>

typedef struct mme_play_info {
uint64_t        fid;
uint64_t        msid;
uint32_t        storage_type;
uint32_t        ftype;
uint32_t        playmode;
uint32_t        slottype;
uint32_t        tracknum;
uint32_t        titlenum;
    uint32_t        audio_index;
    uint32_t        support;
    uint32_t        reserved;
    uint64_t        mscap;
    uint32_t        reserved;
    uint64_t        offset;
} mme_play_info_t;
```

Description:

The structure **mme_play_info_t** carries information about the currently playing track. The function *mme_play_get_info()* uses this structure to deliver information about the state of a playback operation.

Member	Type	Description
fid	**uint64_t**	The track or file ID.
msid	**uint64_t**	The mediastore ID.
storage_type	**uint32_t**	The type of mediastore. See MME_STORAGETYPE_* in this reference.
ftype	**uint32_t**	The type of media track or file. See MME_FTYPE_* in this reference.
playmode	**uint32_t**	The playmode of the track session (library or file). See MME_FORMAT_* and MME_PLAYMODE_* in this reference.
slottype	**uint32_t**	The slot type of the current track or file. See MME_SLOTTYPE_* in this reference.
tracknum	**uint32_t**	The track number of the current track or file.
titlenum	**uint32_t**	The title or group number of the CD, DVD-video or DVD-audio.

continued...

Member	Type	Description
audio_index	**uint32_t**	The audio index of the track on a DVD. It is the same as the *audio_index* filed in the **library**.
support	**uint32_t**	A bitmask flag indicating the functionality supported by the current playing track. See "Play *support* flag" below.
reserved	**uint32_t**	Reserved for future use..
mscap	**uint64_t**	A bitmask with the mediastore capabilities. Values are defined by the MME_MSCAP_* constants.
offset	**uint64_t**	The current offset in the track session. Offsets are zero-based

For information about storage types, see MME_STORAGETYPE_* in this reference.

Play *support* flag

The *support* member of **mme_play_info_t** is a bitmask flag indicating the functionality supported by the current track or file, and the device on which track or file is located:

- MME_PLAYSUPPORT_NAVIGATION — the current track is navigable. Use the function *mme_button()* to allow the end-user to control navigation of the track.
- MME_PLAYSUPPORT_DEVICE_TRACKSESSION — the device supports it own track session management. An example of this functionality is an iPod running in serial mode. Rather than issue *mme_next()*, issue *mme_button(NEXT)* to move to the next track.
- MME_PLAYSUPPORT_VIDEO — the current track has video.
- MME_PLAYSUPPORT_AUDIO — the current track has audio.
- MME_PLAYSUPPORT_REPEATRANDOM — the device track session supports repeat and random. An example of this sort of device is an iPod operating in serial mode.

Classification:

QNX Multimedia

See also:

mme_play_get_info(), *mme_button()*, MME_FTYPE_*, MME_FORMAT_* and MME_PLAYMODE_*, MME_SLOTTYPE_*, MME_STORAGETYPE_*

mme_play_offset()

Start playback at the specified offset in a track session

Synopsis:

```
#include <mme/mme.h>

int mme_play_offset( mme_hdl_t *hdl,
                     int offset,
                     uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
offset	The 0-based offset in the track session at which to start playback.
flags	For future use.

Library:

mme

Description:

The function *mme_play_offset()* starts playback at the specified offset in a track session (the offset in the **trksessionview** table). A value of 0 for the *offset* starts playback of the first track in the track session. Once started, playback continues through to the end of the track session.

Note the following about using *mme_play_offset()*:

- The client application must create and set a track session before using *mme_play_offset()*, just as it does for *mme_play()*.
- A call to *mme_play_offset()* while playback is underway will stop playback and restart it at the specified offset:
  - If the track currently playing is part of a device track session (*mme_play_get_info()* reports MME_PLAYSUPPORT_DEVICE_TRACKSESSION), the MME applies the offset to the *device* tracksession.
  - In all other cases, the MME applies the offset to the MME tracksession. For more information about MME and device track sessions, see “Playing media on iPods” in the chapter Working with External Devices of the *MME Developer’s Guide*.

You need to be playing the iPod before you can jump to an index into its tracksession. You can use *mme_play_get_info()* to confirm that you are playing an iPod: if you are *not* the iPod, *mme_play_get_info()* will *not* report MME_PLAYSUPPORT_DEVICE_TRACKSESSION.

- Random and repeat modes do not change the behavior of *mme_play_offset()*:
  - If random mode is on, playback starts at the specified offset in the random order track session, and continues from that point.
  - If repeat mode is on for the track session, playback repeats through the track session until it is stopped.

  In other words, if the value of *offset*is 1 and:
  - the tracks in a sequential track session are 4, 5, 6, then playback starts with track 5.
  - the tracks in a random track session are 6, 4, 5, then playback starts with track 4.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), mme_trksession_set_files(), mme_trksessionview_readx()

mme_play_resume_msid()

Resume playback of a track session on a specified mediastore.

Synopsis:

```
#include <mme/mme.h>

int mme_play_resume_msid( mme_hdl_t *hdl,
                          uint64_t msid );
```

Arguments:

hdl An MME connection handle.

msid The ID of the mediastore for which to resume playback.

Library:

mme

Description:

The function *mme_play_resume_msid()* resumes playback of a track session on a specified mediastore at the point *mme_trksession_save_state()* saved the track session's state.

For devices, such as iPods, where the device itself maintains state knowledge:

- the function *mme_play_resume_msid()* creates a new track session and resumes playback where indicated by the device's memory.
- calling *mme_play_resume_msid()* when the iPod device is in a stopped state will not resume playback, because a stopped iPod has no active track session that can be resumed.
- after a call to *mme_play_resume_msid()*, you should wait for the MME_EVENT_PLAYSTATE event with the *playstate* set to MME_PLAYSTATE_PLAYING before querying the device or setting the random and repeat modes.

For more information, see the *MME Developer's Guide*:

- "Stopping and resuming playback" in the chapter Playing Media
- "Using random and repeat modes on iPods" in the chapter Working with iPods

Events

The function *mme_play_resume_msid()* delivers the following event:

- MME_EVENT_PLAYSTATE — the function has completed work.

Blocking and validation

This function blocks on **qdb**. In asynchronous mode, it attempts to validate the request (make sure the request makes sense) before releasing the caller.

Returns:

≥**0**	Success: the MME resumed playback of the track session for the mediastore.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_get_info(), mme_play_resume_msid()
mme_trksession_resume_state() mme_trksession_save_state()

mme_play_set_output_attr()

Set the attributes for an output device

Synopsis:

```
#include <mme/mme.h>

int mme_play_set_output_attr( mme_hdl_t *hdl,
                              uint64_t outputdeviceid,
                              mme_output_attr_t *attr );
```

Arguments:

hdl	An MME connection handle.
outputdeviceid	The ID of the output device on which to set attributes.
attr	A pointer to a structure with the output device attributes. See **mme_output_attr_t** in this reference.

Library:

mme

Description:

The function *mme_play_set_output_attr()* sets the output attributes for the specified output device. These attributes are carried in the data structure **mme_output_attr_t** described in this reference.

To apply the same attributes to all output devices attached to a control context, set *outputdeviceid* to 0. The MME will iterate through all attached output devices and apply the values specified in **mme_output_attr_t** to them.

Events

This function delivers MME_EVENT_OUTPUTATTRCHANGE with the ID of the output device where the change occured, in **mme_event_data_t.value**.

Blocking and validation

This function validates the output device ID, and behaves as follows, depending on whether the MME is currently playing a track:

- Playing — if the MME connection is asynchronous, this function returns before updating **io-media**, because **io-media** communicates with hardware, which has different response times, making it impossible for the MME to know how long it will take to return.
- Not playing — behaves synchronously: fully validating, updating a cache, not hardware.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_get_output_attr(), mme_output_set_permanent()

mme_play_set_speed()

Sets playback speed and direction (forward, reverse, pause) for tracks

Synopsis:

```
#include <mme/mme.h>

int mme_play_set_speed( mme_hdl_t *hdl,
                        int speed );
```

Arguments:

hdl An MME connection handle.

speed The playback speed to set for the current track, expressed in units of 1/1000 of normal speed.

Library:

mme

Description:

The function *mme_play_set_speed()* sets the playback speed for the current track or file, including forward, reverse and pause.

The playback speed is expressed in units of 1/1000 of normal speed: 1000 means normal speed, 2000 means double speed, etc. Positive values mean forward, negative values mean reverse, and zero means pause. Values between 0 and 1000 are slow speed playback.

- The requested speed can't be guaranteed for all devices. The graph used to play the track will select the supported speed closest to the one requested. The client application should use *mme_play_get_status()* to get the actual playback speed.
- During fast forward or reverse, an iPod continuously increases speed until it reaches the beginning or end of a track, at which time it resets to normal speed.

Events

MME_EVENT_TIME when the function has completed work.

Blocking and validation

This function verifies that the requested time position is valid, and blocks until it has advanced playback to this time position.

Returns:

≥`0`	Success: the playback speed was set.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), *mme_play_get_info()*, *mme_play_get_output_attr()*, *mme_play_get_speed()*, *mme_play_get_status()*, *mme_play_set_output_attr()*

mme_play_set_zone()

Set the zone ID used by a control context

Synopsis:

```
#include <mme/mme.h>

int mme_play_set_zone( mme_hdl_t *hdl,
                       uint64_t zoneid );
```

Arguments:

hdl An MME connection handle.

zoneid The ID of the output zone to be used by the current control context.

Library:

mme

Description:

The function *mme_play_set_zone()* sets the output zone to be used by the current control context. For more information about zones, see *mme_zone_create()*.

Events

None delivered.

Blocking and validation

This function is fully validating and runs to completion.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_output_set_permanent(), mme_play_attach_output(), mme_play_detach_output(), mme_play_set_zone(), mme_zone_create(), mme_zone_delete()

mme_play_status_t

Play status information

Synopsis:

```
#include <mme/types.h>

typedef struct _mme_play_status {
    mme_time_t  time_info;
    uint32_t    playstate;
    uint32_t    buffer;
} mme_play_status_t;
```

Description:

The structure **mme_play_status_t** provides a snapshot of the current playback status, including total play time and play time elapsed. It includes at least the members described in the table below.

Member	Type	Description
time_info	**mme_time_t**	Time information about the current track or file. See **mme_time_t** in this reference.
playstate	**mme_playstate_t**	The current MME playstate. See **mme_playstate_t** in this reference.
buffer	**mme_buffer_status_t**	The current playback buffer status. See **mme_buffer_status_t**.

Classification:

QNX Multimedia

See also:

mme_buffer_status_t, **mme_time_t**, **mme_playstate_t**, **mme_playstate_speed_t**

Synopsis:

```
#include <mme/interface.h>

#define MME_PLAYLIST_*
```

Description:

The MME_PLAYLIST_* constants define values used in the **`playlists`** table,:

- MME_PLAYLIST_MODE_*
- MME_PLAYLIST_OWNER_*

See also the MME_PLAYLIST_FLAGS_PLAYLIST_ENTRY and MME_PLAYLIST_RESOLVE_* constants used by *mme_playlist_item_get()*.

MME_PLAYLIST_MODE_*

The MME_PLAYLIST_MODE_* constants identify the type of playlist. The MME updates the *mode* field in the **`playlists`** table with the value identifying the playlist mode.

Constant	Value	Description
MME_PLAYLIST_MODE_LIBRARY	0	The playlist is on a mediastore.
MME_PLAYLIST_MODE_GENERATED	1	The playlist has been created by the user.

MME_PLAYLIST_OWNER_*

The MME_PLAYLIST_OWNER_* constants identify the owner of a playlist. The MME updates the *ownership* field in the **`playlists`** table with the value identifying the playlist owner.

Constant	Value	Description
MME_PLAYLIST_OWNER_MME	0	The playlist is owned by the MME.
MME_PLAYLIST_OWNER_DEVICE	1	The playlist is owned by an external device, such as an iPod.
MME_PLAYLIST_OWNER_USER	2	The playlist is owned by the user, who created the playlist.

Classification:

QNX Multimedia

See also:

mme_playlist_close(), *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_close()

Close a playlist

Synopsis:

```
#include <mme/playlist.h>

int mme_playlist_close( mme_playlist_hdl_t *hdl );
```

Arguments:

hdl An MME playlist connection handle returned by *mme_playlist_open()*.

Library:

mme

Description:

The function *mme_playlist_close()* closes the playlist opened with the connection handle referenced by *hdl*.

Events

None returned

Blocking and validation

This function validates the playlist connection handle and does not block.

Returns:

0 Success: the ID of the synchronization operation.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **mme_playlist_hdl_t**,

mme_playlist_item_get(), mme_playlist_items_count_get(), mme_playlist_open(), mme_playlist_position_set(), mme_playlist_set_statement(), mme_playlist_sync()

Synopsis:

```
#include <mme/playlist.h>

int mme_playlist_create( mme_hdl_t *hdl,
                         uint64_t msid,
                         const char *name,
                         uint64_t *plid );
```

Arguments:

hdl	An MME connection handle.
msid	The ID for the mediastore from which the playlist will be made. If the mediastore is pruned, the playlist will be deleted. Set the mediastore ID to 0 (zero) to prevent pruning of the mediastore.
name	The name of the new playlist.
plid	The ID for the new playlist.

Library:

mme

Description:

The function *mme_playlist_create()* creates a new playlist from a mediastore. It adds a playlist entry to the table **playlists** and the playlist data to the **playlistdata** table. It does not write to the **playlistdata_custom** table, or any other ***_custom** tables; these remain the responsibility of the client application.

Events

None delivered.

Blocking and validation

This function performs no validations, and runs to completion.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_delete()

Delete a specified playlist

Synopsis:

```
#include <mme/mme.h>

int mme_playlist_delete( mme_hdl_t *hdl,
                         uint64_t *plid );
```

Arguments:

hdl An MME connection handle.

plid The ID of the playlist to be deleted.

Library:

mme

Description:

The function *mme_playlist_delete()* deletes a playlist from the **playlists** table, and its data from the **playlistdata** table.

This function does not delete custom playlists in the **playlistdata_custom** table. Custom playlists must be deleted manually.

The following example provided in **mme_connect.sql** shows how to create triggers to delete entries from the **playlistdata_custom** table when the client application calls *mme_playlist_delete()* to delete a playlist:

```
CREATE TEMP TRIGGER playlistdata_custom_delete DELETE ON playlists
    BEGIN
        DELETE FROM playlistdata_custom WHERE plid=OLD.plid;
    END;
```

Events

None delivered.

Blocking and validation

This function runs to completion.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

This function is no yet fully implemented, and returns an ENOSUP error if it is called.

Synopsis:

```
#include <mme/playlist.h>

int mme_playlist_generate_similar( mme_hdl_t *hdl,
                                   const char *name
                                   uint64_t fid,
                                   uint64_t msid,
                                   unsigned max_entries,
                                   uint32_t flags,
                                   uint32_t *plid );
```

Arguments:

hdl	An MME connection handle.
name	A pointer to a text name to display for the new playlist.
fid	The ID of the file to use as a seed for the new playlist.
msid	The ID of the mediastore from which to select tracks to place in the playlist. See "Playlists and mediastores" below.
max_entries	The maximum number of entries that can be put in the new playlist.
flags	For future use.
plid	The playlist ID of the new playlist.

Library:

mme

Description:

The function *mme_playlist_generate_similar()* generates a playlist from files similar to the seed file.

Playlists and mediastores

The *msid* argument determines which mediastores *mme_playlist_generate_similar()* uses to generate a playlist. Possible values and behaviors are as follows:

>0 Build a playlist from tracks on the specified mediastore.

If the MME prunes the mediastore from its database, it also prunes the playlist.

=0 Build a playlist from tracks on all active mediastores.

The client application is responsible for pruning the playlist when it is no longer needed; the MME does *not* prune the playlist from the database, because it is not associated with a specific mediastore.

Events

None delivered.

Blocking and validation

This function validates the mediastore ID, and runs to completion.

Returns:

`>0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_hdl_t

The playlist API handle

Synopsis:

```
#include <mme/playlist.h>

struct mme_playlist_hdl;
typedef struct mme_playlist_hdl mme_playlist_hdl_t;
```

Description:

The structure **mme_playlist_hdl_t** is used for playlist session control. One handle is used for each playlist opened.

Classification:

QNX Multimedia

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_item_get()

Get an entry from a playlist

Synopsis:

```
#include <mme/playlist.h>

int mme_playlist_item_get( mme_playlist_hdl_t *hdl,
                           uint32_t *flags,
                           char *buffer,
                           size_t length );
```

Arguments:

hdl	A playlist connection handle.
flags	A pointer to flags to control the operation. See "Flags" below.
buffer	A pointer to a buffer
length	The length of the buffer, in bytes; may be 0.

Library:

mme

Description:

The function *mme_playlist_item_get()* retrieves the playlist entry at the position specified by *mme_playlist_position_set()*, and places it in the buffer referenced by *buffer*.

Successful completion (return value≥0) of a call to *mme_playlist_item_get()* does not mean that the function successfully read in the playlist entry. If the returned value is great than the allocated buffer length (*length*), you must increase the buffer length to at least the returned value and call the function again to read in the entry. Alternately, you can call *mme_playlist_item_get()* with the *length* argument set to 0 to get the playlist entry length, set the buffer size to the returned value, then call the function again.

Flags

The *flags* argument is used both to:

- Pass instructions to *mme_playlist_item_get()*: when calling *mme_playlist_item_get()*, set the *flags* argument to MME_PLAYLIST_RESOLVE_PLAYLIST_ENTRY to have the function convert the playlist entry to a file.
- Return information about the retrieved playlist entry: the entry is either unconverted (MME_PLAYLIST_FLAGS_PLAYLIST_ENTRY) or convereted into a file (MME_PLAYLIST_FLAGS_PLAYLIST_FILE).

MME_PLAYLIST_FLAGS_*

The MME_PLAYLIST_FLAGS_* constants identify the type of item that has been retreived by a call to *mme_playlist_item_get()*.

Constant	Value	Description
MME_PLAYLIST_FLAGS_PLAYLIST_ENTRY	0x00000001	The item is an unconverted entry from a playlist.
MME_PLAYLIST_FLAGS_PLAYLIST_FILE	0x00000002	The item is a playlist entry that has been converted to a real file.

MME_PLAYLIST_RESOLVE_*

The MME_PLAYLIST_RESOLVE_* constants determine how to process a playlist item retrieved with *mme_playlist_item_get()*.

Constant	Value	Description
MME_PLAYLIST_RESOLVE_PLAYLIST_ENTRY	0x00000001	Convert the entry to a real file before returning it.

Events

None delivered.

Blocking and validation

This function performs no validations, and runs to completion.

Returns:

`>0` Success: the length of the playlist entry, in bytes, even if the buffer is too short for the entry.

`0` Success, but the end of the playlist has been reached.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_item_get()*

mme_playlist_items_count_get()

Get the number of items in a playlist

Synopsis:

```
#include <mme/playlist.h>

int mme_playlist_items_count_get( mme_playlist_hdl_t *hdl,
                                  int *items );
```

Arguments:

hdl An playlist connection handle.

items The number of items in the playlist.

Library:

mme

Description:

The function *mme_playlist_items_count_get()* gets the number of items in the currently open playlist. This number can be 0, greater than 0, or -1. If the value is -1, the playlist has no fixed length.

Events

None delivered.

Blocking and validation

This function performs no validations, and runs to completion.

Returns:

0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_open()

Open a playlist

Synopsis:

```
#include <mme/playlist.h>

mme_playlist_hdl_t mme_playlist_open( mme_hdl_t *hdl,
                                      uint64_t plid,
                                      uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
folderid	The ID of the playlist to open.
flags	For future use.

Library:

mme

Description:

The function *mme_playlist_open()* returns a handle to be used to work with a playlist. After calling *mme_playlist_open()*, you can use other *mme_playlist_*()* functions to find and extract entries from the opened playlist.

This function can only open a playlist if a playlist synchronization (PLSS) plugin able to process the playlist is available. If no PLSS plugin for the playlist is available, *mme_playlist_open()* fails.

Events

None delivered.

Blocking and validation

This function validates the playlist ID, and runs to completion.

Returns:

An initialized **mme_playlist_hdl_t**, or NULL if an error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_position_set()

Set the current position in a playlist

Synopsis:

```
#include <mme/mme.h>

int mme_playlist_position_set( mme_playlist_hdl_t *hdl,
                               unsigned position );
```

Arguments:

hdl	A playlist connection handle.
position	The position to set in the playlist to.

Library:

mme/playlist

Description:

The function *mme_playlist_position_set()* sets a position in the current playlist. After calling this function, you can call *mme_playlist_item_get()* to retrieve the item from the position set.

Events

None delivered.

Blocking and validation

This function performs no validations, and runs to completion.

Returns:

0	Success: the ID of the synchronization operation.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_set_statement()*, *mme_playlist_sync()*

mme_playlist_set_statement()

Set the SQL statement to create a playlist

Synopsis:

```
#include <mme/mme.h>

int mme_playlist_set_statement( mme_hdl_t *hdl,
                                uint64_t *plid,
                                const char *sql );
```

Arguments:

hdl An MME connection handle.

plid The ID of the playlist.

sql A pointer to the SQL statement used to retrieve the file IDs of the files or tracks for the playlist.

Library:

mme

Description:

The function *mme_playlist_set_statement()* sets the SQL statement to use when retrieving the files to create a playlist.

Events

Blocking and validation

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **`mme_playlist_hdl_t`**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_sync()*,

Synchronize a specified playlist

Synopsis:

```
#include <mme/mme.h>

int mme_playlist_sync( mme_hdl_t *hdl,
                       uint64_t plid,
                       uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
folderid	The ID of the playlist to synchronize.
flags	For future use.

Library:

mme

Description:

The function *mme_playlist_sync()* synchronizes the specified playlist. When it completes the synchronization operation it delivers either MME_EVENT_MS_SYNCCOMPLETE for a successfully synchronization, or MME_EVENT_SYNCABORTED for an unsuccessful synchronization.

Events

The function *mme_playlist_sync()* may deliver any event of the class MME_EVENT_CLASS_SYNC, and any of the MME_SYNC_ERROR_* error events.

Blocking and validation

This function validates the synchronization request and does not block.

Returns:

>0	Success: the ID of the synchronization operation.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_PLAYLIST_*, *mme_playlist_close()*, *mme_playlist_create()*, *mme_playlist_delete()*, *mme_playlist_generate_similar()*, **mme_playlist_hdl_t**, *mme_playlist_item_get()*, *mme_playlist_items_count_get()*, *mme_playlist_open()*, *mme_playlist_position_set()*, *mme_playlist_set_statement()*, *mme_resync_mediastore()*

mme_playstate_speed_t

Playback state and speed

Synopsis:

```
#include <mme/types.h>
typedef struct mme_playstate_speed {
    uint32_t    playstate;
    int32_t     speed;
} mme_playstate_speed_t;
```

Description:

The structure **mme_playstate_speed_t** carries information about state and speed of playback. The MME uses this structure with the event MME_EVENT_PLAYSTATE to deliver information about the state of a playback operation.

Member	Type	Description
playstate	**uint32_t**	The current play state. See **mme_playstate_t**
speed	**int32_t**	The current playback speed, expressed in units of 1/1000 of normal speed: 1000 means normal speed, 2000 means double speed, etc. Positive values mean forward, negative values mean reverse, and zero means pause. Values between 0 and 1000 are slow speed playback.

Classification:

QNX Multimedia

See also:

mme_play_get_status(), **mme_playstate_t**, **mme_play_status_t**

Synopsis:

```
#include <mme/types.h>

typedef enum mme_playstate {
    ...
} mme_playstate_t;
```

Description:

The enumerated type **mme_playstate_t** defines the values used to describe playback states. Its values include:

- MME_PLAYSTATE_UNKNOWN (0)
- MME_PLAYSTATE_ERROR (1)
- MME_PLAYSTATE_PLAYING (2)
- MME_PLAYSTATE_PAUSED (3)
- MME_PLAYSTATE_FASTFWD (4)
- MME_PLAYSTATE_FASTREV (5)
- Not used. (6)
- MME_PLAYSTATE_STOPPED (7)
- MME_PLAYSTATE_SLOWFWD (8)
- MME_PLAYSTATE_SLOWREV (9)

Classification:

QNX Multimedia

See also:

mme_play_get_status(), **mme_playstate_speed_t**, **mme_play_status_t**

Synopsis:

```
#include <mme/mme.h>

int mme_prev( mme_hdl_t *hdl );
```

Arguments:

hdl The MME handle for the control context playing the track session on which you want to skip to the previous track.

Library:

mme

Description:

The function *mme_prev()* skips to the previous title in the currently playing track session. The previously played track is obtained from the **trksessionview** table.

Effect of play modes on behavior

The behavior of *mme_prev()* is affected by the play modes set for the specified control context (sequential versus random, and repeat versus no repeat).

If sequential mode is set, the file ID of the previous track in the track session is in the previous row in the *sequentialid* column of the **trksessionview** table. If random mode is set, the file ID of the previous track in the track session is in the *randomid* column of the **trksessionview** table.

Effect of repeat mode on the first track of a session

When the first track in the track session is playing, the result of calling *mme_prev()* depends on whether the repeat mode is set.

If repeat mode is off, when it has reached the beginning of the track session (or, when random mode is set, when all songs in the track session have been played), *mme_next()* sets *errno* to ENODATA .

If repeat mode is on:

- if sequential mode is set, the MME plays the first track in the track session, as determined by the *sequentialid* column in the **trksessionview** table.
- if random mode is set, the MME plays the first track in the track session, as determined by the *randomid* column in the **trksessionview** table.

Working with an iPod device

iPod devices manage their own track sessions. To move to the next or previous track in an iPod track session, call the *mme_button()* function with **mm_button_t** set to MM_BUTTON_NEXT or MM_BUTTON_PREV, as required.

Events

This function may deliver any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

This function verifies that the *fid* is valid. Does not verify that the file exists, or that it is playable.

This function blocks on control contexts. If *mme_prev()* is called and another function is called before *mme_prev()* returns, the second function blocks on **io-media** until *mme_prev()* returns. If there are no other pending calls, *mme_prev()* returns without blocking on **io-media**.

Returns:

≥0 Success: *errno* set to ENODATA indicates that there are no more tracks to play.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_next(), *mme_setrandom()*, *mme_setrepeat()*

Synopsis:

```
#include <mme/mme.h>

int mme_register_for_events( mme_hdl_t *hdl,
                             mme_event_class_t event_class,
                             struct sigevent *event );
```

Arguments:

hdl	The MME connection handle.
event_class	The MME event class or classes for which the client application wants to register or unregister.
event	The event to have delivered when it is received. To unregister for the specified class set *event* to NULL.

Library:

mme

Description:

The function *mme_register_for_events()* allows the client application to determine the events it wants to receive from the MME.

Register for events

The MME does not deliver events to a client application unless it is specifically instructed to do so. To receive events from the MME, a client application must register for events after connecting to the MME, specifying the class or classes of events it wants to receive.

The client application must register after each connection. This feature allows the client application to register different different classes of events for connections. For example, a connection used to handle synchronizations can register for synchronization events, but not for playback events, because it will never call functions that deliver playback events.

Each event class has a different sigevent. When it has registered for an event class, the client application has told the MME which sigevents it wants to receive. When it has a relevant event, the MME will:

- place it in its event queue
- send the **sigevent** automatically to the client application.

The client application can then decide from the sigevent if it needs to see the associated event. When it needs to see events, the client application can use the function *mme_get_event()* to have them delivered from the MME's event queue.

Unregister for events

To stop receiving a class of events, the client application must unregister for that event class. To unregister for an event class, call the function *mme_register_for_events()* with the *event_class* set to the event class for which you want to stop receiving events, and the argument *event* set to NULL.

If the client application has registered for several or all event classes, it can unregister for any event class without affecting the registration for the other event classes. For example:

```
mme_register_for_events( hdl, MME_EVENT_CLASS_ALL, &event );

// Do some work here.

mme_register_for_events( hdl, MM_EVENT_CLASS_COPY, NULL);
```

MME event classes

mme_event_classes_t defines the different MME event classes as bitmasks:

MME_EVENT_CLASS_PLAY
: Playback events.

MME_EVENT_CLASS_SYNC
: Synchronization events.

MME_EVENT_CLASS_COPY
: Copying and ripping events.

MME_EVENT_CLASS_GENERAL
: Events not specified in the other classes.

MME_EVENT_CLASS_ALL
: All events.

The MME event classes are bitmasks. They can be used together with an OR operator to register for several events at once. For example, to register for *playback* and *synchronization* events call the function *mme_register_for_events()* as follows:

```
mme_register_for_events( hdl,
                         MME_EVENT_CLASS_PLAY | MME_EVENT_CLASS_SYNC,
                         event);
```

For more information about events, see the chapter MME Events and following.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect(), *mme_disconnect()*, *mme_get_event()*, MME Events

mme_resync_mediastore()

Re-synchronize a mediastore

Synopsis:

```
#include <mme/mme.h>

int mme_resync_mediastore( mme_hdl_t *hdl,
                           uint64_t msid,
                           uint64_t folderid,
                           uint32_t options );
```

Arguments:

hdl An MME connection handle.

msid The ID of the mediastore to resynchronize.

folderid Specifies the folder to synchronize. A value of 0 means synchronize all folders.

options A mask that sets synchronization options. The options can be any combination of:

- MME_SYNC_OPTION_CANCEL_CURRENT — not used by *mme_resync_mediastore()*. See *mme_sync_directed()*.
- MME_SYNC_OPTION_CLR_INV_COPIED — set to 0 (zero) all invalid *copied_fid* values in the **library** table.
- MME_SYNC_OPTION_PASS_FILES — synchronize files (perform first pass synchronization).
- MME_SYNC_OPTION_PASS_METADATA — synchronize metadata (perform second pass synchronization).
- MME_SYNC_OPTION_PASS_PLAYLISTS — synchronize playlists (perform third pass synchronization).
- MME_SYNC_OPTION_PASS_ALL — synchronize files, metadata, and playlists.
- MME_SYNC_OPTION_RECURSIVE — perform a recursive synchronization; *mme_resync_mediastore()* always assumes that this flag is set.

Library:

mme

Description:

The function *mme_resync_mediastore()* attempts to start synchronization of a mediastore. It returns immediately, with synchronization continuing in the background.

When a particular pass is specified, if that pass was previously marked as complete in the MME database, the MME first marks that pass as not complete, then attempts the new synchronization. Any previously completed synchronization passes that are not being redone are left untouched.

CAUTION: A clean up of invalid *copied_id* fields can take a long time. Use the MME_SYNC_OPTION_CLR_INV_COPIED flag judiciously — *only* when synchronizing after deleting media files from your database.

Events

The function *mme_resync_mediastore()* may deliver any event of the class MME_EVENT_CLASS_SYNC, and any of the MME_SYNC_ERROR_* error events.

Blocking and validation

This function verifies that the *msid* and *folderid* are valid. It returns, then requests a synchronization in the background at the earliest possible time. If all synchronization threads are busy, this request is queued until a synchronization thread becomes available.

See the chapter Configuring Synchronization in the *MME Configuration Guide*.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), *mme_playlist_sync()*, *mme_setpriorityfolder()*, *mme_sync_cancel()*, *mme_sync_directed()*, *mme_sync_file()*, *mme_sync_get_msid_status()*, *mme_sync_get_status()*

mme_rmtrksession()

Remove a track session from the database

Synopsis:

```
#include <mme/mme.h>
int mme_rmtrksession( mme_hdl_t *hdl,
                      uint64_t trksessionid );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The ID for the track session you want to remove.

Library:

mme

Description:

The function *mme_rmtrksession()* removes the specified track session from the **trksessions** table in the MME library. It also removes references to the specified track session from these other tables:

- **controlcontexts** table — if the removed track session is the currently playing track session for the control context, the *trksessionid* field for the control context is set to 0
- **mediastores** — if the removed track session was the last played track session for this mediastore, the *trksessionid* field for the control context is set to 0

You can get the current track session for a control context by calling *mme_trksession_get_info()*.

Events

None delivered.

Blocking and validation

This function blocks on control contexts. It fully validates data; all arguments are checked before the call returns.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_get_info(), mme_trksession_resume_state(), mme_set_msid_resume_trksession(), mme_newtrksession(), settrksession()

mme_seek_title_chapter()

Seek to a specified title and chapter on a track or mediastore

Synopsis:

```
#include <mme/mme.h>

int mme_seek_title_chapter( mme_hdl_t hdl,
                            uint64_t title,
                            uint64_t chapter );
```

Arguments:

hdl	An MME connection handle.
title	The title from which to start playback.
chapter	The chapter from which to start playback.

Library:

mme

Description:

The function *mme_seek_title_chapter()* seeks to a specified title and chapter on a track or mediastore so that playback can begin from that point. This function can only be used if the MME_PLAYSUPPORT_NAVIGATION flag is set in the *support* member of the structure **mme_play_info_t**.

To start playback from a specific title and chapter:

1. Create a track session with the mediastore file ID (*fid*) for the entire DVD.
2. Set the track session.
3. Call *mme_play()* to start playback.
4. Once the navigator is active, call *mme_seek_title_chapter()* to seek to the desired title and chapter on the DVD.

To get information about titles and chapters on a playing track, call the function *mme_get_title_chapter()*.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Examples:

Below is a code snippet that illustrates how to seek to a specific title (1) and chapter (5).

```
uint64_t title = 1, chapter = 5;

rc = mme_seek_title_chapter(mmehdl, title, chapter);
if (rc == EOK) {
    printf( "Seeking to title %lld chapter %lld", title, chapter);
} else {
    printf( "Seek to title %lld chapter %lld failed, %s", title, chapter, strerror(errno))
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_get_title_chapter(), *mme_play()*, *mme_play_bookmark()*, *mme_play_get_info()*, **mme_play_info_t**, *mme_seektotime()*

mme_seektotime()

Seek to a time in a playing track

Synopsis:

```
#include <mme/mme.h>

int mme_seektotime( mme_hdl_t *hdl,
                    int time );
```

Arguments:

hdl An MME connection handle.

time The time you want to seek to, in milliseconds.

Library:

mme

Description:

The function *mme_seektotime()* seeks to a specific time (expressed in milliseconds from the start of the track) in the current track. The track must be playing for the seek to work.

If *time* is greater than the total time for the currently playing track, behavior varies, depending on the media, as follows:

- DVD-audio — return to the beginning of the current track
- DVD-video — seek to the requested time in the title
- all other media — seek to the end of the current track

Events

None delivered.

Blocking and validation

This function blocks on control contexts. If *mme_seektotime()* is called and another function is called before *mme_seektotime()* returns, the second function blocks on **io-media** until *mme_seektotime()* returns. If there are no other pending calls, *mme_seektotime()* returns without blocking on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect(), *mme_next()*, *mme_play()*

mme_set_api_timeout()

Set the time period on the unblocking timer

Synopsis:

```
#include <mme/mme.h>

int mme_set_api_timeout( mme_hdl_t *hdl,
                         uint32_t *milliseconds );
```

Arguments:

hdl	An MME connection handle.
milliseconds	The time, in milliseconds, to wait before unblocking the client. Set to 0 to disable.

Library:

mme

Description:

The function *mme_set_api_timeout()* sets, for the current control context, the amount of time, in milliseconds, the MME will wait before unblocking the client application when it is blocked by calls to the MME.

If *mme_set_api_timeout()* is set, API calls that are blocked beyond the set timeout period will unblock the client, returning early with the *errno* set to EINTR.

For information about how to confirm the cause of an EINTR error, see *mme_get_api_timeout_remaining()*.

The MME's default configuration is to disable unblocking capabilities, which disables *mme_set_api_timeout()*. To enable the MME's unblocking capability, set the **<Unblock>** configuration element attribute to "true".

CAUTION: The MME connection handle, **mme_hdl_t**, is not thread safe; only one instance can be used at a time. This limitation means that *mme_set_api_timeout()* can *not* be called concurrently with another function call: you can call *mme_set_api_timeout()* at any time to configure the behavior of *future* calls to the MME API, but you can't use *mme_set_api_timeout()* to force the return of a call that has already been made.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set). Errno is set.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_get_api_timeout_remaining()

mme_set_debug()

Set MME debug settings

Synopsis:

```
#include <mme/mme.h>

int mme_set_debug( mme_hdl_t *hdl,
                   uint8_t verbose,
                   uint8_t debug );
```

Arguments:

hdl	An MME connection handle.
verbose	The verbosity setting for the MME.
debug	The debug setting for the MME.

Library:

mme

Description:

The function *mme_set_debug()* sets the MME verbosity and debug levels. It can be called at any time. Debug and verbosity levels range from 0 (zero) to 10, with 0 meaning “turned off” and 10 providing the most detailed information. These levels are equivalent to the **mme** start up options **-v** and **-D**. See also **mme** in the *MME Utilities Reference*.

When debugging problems, use a higher verbosity level to write more detailed information to the log. The debug setting is usually used only by QNX developers.

CAUTION: The higher the verbosity and debug settings, the more overhead is placed on the system. A production environment should run with verbosity and debug settings of 0 (zero).

Events

None delivered.

Blocking and validation

This function blocks until it completes.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_set_api_timeout(), mme_sync_set_debug()

mme_set_files_permanent()

Set files as permanent (not prunable) or prunable

Synopsis:

```
#include <mme/mme.h>

int mme_set_files_permanent( mme_hdl_t *hdl,
                             bool permanent,
                             const char *fidselect );
```

Arguments:

hdl	An MME connection handle.
permanent	A boolean flag to set the file as permanent “true” (permanent) or “false” (prunable).
fidselect	A SELECT statement to retrieve file or files to be marked.

Library:

mme

Description:

The function *mme_set_files_permanent()* marks specified media files as permanent (not prunable), or prunable. This feature can be used to ensure that files, such as ring tones, are never pruned from the MME’s database. The default setting for files is prunable.

To mark one or more files as either permanent or prunable, call *mme_set_files_permanent()* with a SELECT statement to select the file or files from the **library** table, and the *permanent* argument set to “true” (permanent) or **false** (prunable), as required. This action sets the *permanent* field in the **library** table for the selected file or files. When the MME is pruning its database it will *not* remove files with the *permanent* field set to **true**.

For more information about prune management, see “Database pruning” in the chapter Configuring Device Support and Media Synchronization of the *MME Configuration Guide*.

Events

None delivered.

Blocking and validation

This function doesn’t block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_resync_mediastore()

mme_set_msid_resume_trksession()

Set the track session ID to use when resuming playback of a mediastore.

Synopsis:

```
#include <mme/mme.h>

int mme_set_msid_resume_trksession( mme_hdl_t *hdl,
                                    uint64_t msid );
```

Arguments:

hdl An MME connection handle.

msid The ID of the mediastore to which the track session ID is set.

Library:

mme

Description:

The function *mme_set_msid_resume_trksession()* links a track session with a specific mediastore. The track session ID is used by the function *mme_play_resume_msid()* to resume playback on the the mediastore.

Multiple mediastore IDs can be assigned to the same track session ID.

For more information about stopping an resuming playback of track sessions, see "Stopping and resuming playback" in the chapter Playing Media of the *MME Developer's Guide*.

Events

None delivered.

Blocking and validation

This function blocks on the control context. It performs full validation and runs to completion, returning success or failure.

Returns:

≥0 Success: the MME assigned the *msid* to the *trksessionid*.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_newtrksession(), mme_rmtrksession(), settrksession(). mme_trksession_get_info(), mme_trksession_resume_state(), mme_trksession_save_state()

mme_set_notification_interval()

Set the time interval between updates during playback

Synopsis:

```
#include <mme/mme.h>

int mme_set_notification_interval( mme_hdl_t *hdl,
                                   uint32_t time );
```

Arguments:

hdl An MME connection handle.

time The time interval between updates.

Library:

mme

Description:

The function *mme_set_notification_interval()* configures the MME to deliver the event MME_EVENT_TIME at regular intervals to the client application, when the MME control context to which the client application is connected is playing a file or track.

The argument *time* sets the event delivery period. The default period is 100 milliseconds.

The deliver period remains constant regardless of the speed of the playback. That is, if the period is set to 100, the MME delivers the event MME_EVENT_TIME to the client application every 100 milliseconds. This represents 100 milliseconds of playback time at the regular speed of 1000, but 200 milliseconds of playback time if the playback speed is 2000.

The only exception is if the playback is stopped, in which case the playback speed is 0 and the MME does not deliver the event MME_EVENT_TIME to the client application.

The table below shows some examples of behavior set by *mme_set_notification_interval()*.

time	Playback speed	Time between notifications	Playback time between notifications
100	1000	100 ms	100 ms
100	2000	100 ms	200 ms
100	500	100 ms	50 ms

continued...

time	Playback speed	Time between notifications	Playback time between notifications
200	2000	200 ms	400 ms
100	0 (paused)	no notification sent	n/a

For more information, see *mme_play_set_speed()*.

Limitations of time reporting accuracy

The accuracy and frequency of time updates depends upon the implementation of the **io-media** graphs used to process the media, and on the accuracy and frequency of updates delivered by the underlying drivers and hardware. Graphs should attempt to deliver a timing resolution of 100 milliseconds or better, but this resolution is not always available.

The MME delivers the MME_EVENT_TIME event to the client application only when it receives a time update from the device or driver (through **io-media**). Thus, if, for example, the MME's notification interval to the client application is set to 100 milliseconds, but a driver delivers time position updates to the MME only every 300 milliseconds, the client application will only receive time updates every 300 milliseconds and may see jitter in the time reporting.

Note also that notification intervals are *approximate*. Actual intervals may vary slightly, depending on the behavior of devices and drivers, and the time required for requesting and receiving time updates.

Events

None delivered

Blocking and validation

This function does not make calls to **qdb** or **io-media**. It blocks only at the control context level; that is, it blocks only if other requests are already queued or being processed. It validates that the notification interval is not being set to 0.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play(), mme_play_get_info() mme_play_get_output_attr() mme_play_get_status() mme_play_output_attr() mme_play_set_speed()

mme_setautopause()

Set the autopause mode for a control context

Synopsis:

```
#include <mme/mme.h>

int mme_setautopause( mme_hdl_t *hdl,
                      bool enable );
```

Arguments:

hdl An MME connection handle.

enable The autopause setting. Pass as **true** to turn on autopause mode, and false to turn it off.

Library:

mme

Description:

The function *mme_setautopause()* sets the autopause mode for a control context. Changing the autopause mode for a control context doesn't affect a currently playing track. The change comes into effect for the next track played. By default, autopause mode is off.

The ability to set a control context's default behavior to start tracks in the paused state is particularly useful if you need to perform additional audio processing outside the MME before playing tracks, or if the system needs to change mediastores during playback of a tracksession.

Playback behavior when autopause mode is on

When autopause is turned on, tracks start playback in the paused state. When a track is started in the paused state, the MME delivers a MME_EVENT_PLAYAUTOPAUSED event, and you need to explicitly resume paused tracks with a call to *mme_play_set_speed()* with *speed* set to 1000.

This behavior affects all calls that initiate playback of a track, including:

- *mme_play()*
- *mme_prev()*
- *mme_next()*

Autopause with devices that control their own track sessions

Do *not* set autopause for control contexts with devices, such as iPods and Bluetooth phones, that control their own playback. If you set autopause for control contexts with these devices attached:

- playback for these devices may produce unexpected behavior
- metadata and other track information requested from these devices may be invalid

Autopause with playback pre-queuing

Autopause will *not* take effect if all the following conditions are true:

- the mediastore IDs of the currently playing track and the next track are the same
- **`io-media`** will use the same graph to play the next track as it is using for the currently playing track (the tracks are of the same format)

See also "Playback pre-queuing" in the chapter Configuring Playback of the *MME Configuration Guide*.

Events

This function delivers the event MME_EVENT_AUTOPAUSECHANGED, if it has changed the autopause state for the control context (for example, from "on" to "off", or from "off" to "on"). If *mme_setautopause()* doesn't change the autopause state for the control context, it doesn't deliver an event (for example, if the state was "on" and was set to "on", or the state was "off" and was set to "off").

Blocking and validation

This function blocks on control contexts.

Returns:

`≥0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getautopause(), mme_next(), mme_play(), mme_prev()

mme_setlocale()

Set the preferred language for media with unknown language

Synopsis:

```
#include <mme/mme.h>

int mme_setlocale ( mme_hdl_t *hdl,
                    const char *locale );
```

Arguments:

hdl An MME connection handle.

locale The locale code to set. This is a string containing a 5-character language and region code. This string consists of a 2-character ISO639-1 language code, followed by a "_" character, followed by a 2-character ISO3166-1 alpha-2 region code. See **http://www.loc.gov/standards/iso639-2/php/code_list.php**.

Library:

mme

Description:

The function *mme_setlocale()* sets the preferred language for displaying:

- MME messages, such as "synchronizing"
- metadata labels, such as "Artist", for media for which the language is not known.

The requested language must exist in the database, and the **languages** table must be populated with the appropriate text strings.

This function doesn't set the preferred language for media playback. To specify that setting, use *mme_media_set_def_lang()*.

The current MME implementation uses only the first two characters to extract the language. In the future, this function may set the language used in strings where language sets are available, causing a re-ordering of database tables that are lexicographically collated.

Events

None delivered.

Blocking and validation

This function is fully validating and runs to completion.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getlocale(), *mme_media_get_def_lang()*, *mme_media_set_def_lang()*

mme_set_logging()

Set the verbosity levels for specified logging modules

Synopsis:

```
#include <mme/mme.h>

int mme_set_logging( mme_hdl_t *hdl,
                     const char *name,
                     uint8_t level,
                     uint8_t flags );
```

Arguments:

hdl	An MME connection handle.
name	A pointer to a string with the name of the logging module for which log levels are to be set. To set levels for all modules, set the string to NULL. See the "Description" below.
verbose	The new log verbosity level to use for the specified modules. See "Logging modules" below.
flags	Flags that configure logging behavior. See "Logging flags" below.

Library:

mme

Description:

The function *mme_set_logging()* sets the verbosity levels for specified MME logging modules. You can set verbosity levels as required for individual modules or for all modules, as required, by placing the appropriate strings in the buffer referenced by the *name* argument.

Logging modules

The strings that identify **mme** logging modules include:

String	Module
imgprc	image processing module
mdi	metadata interface module
mdp	metdata plugin module
pl	playlist module
sync	synchronization module

continued...

String	Module
`mme`	all other modules

The above list is not definitive. The logging modules may change. To find out what logging module strings are valid, call *mme_get_logging()* with the string referenced by the *name* argument set to NULL.

Logging flags

The logging flags are bit masks that configure logging behavior:

Value	Behavior
1	Also write anything logged to standard output.
2	Write timing logs.

Events

None delivered.

Blocking and validation

This function doesn't perform any validations, and doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_get_logging()

mme_setpriorityfolder()

Set a priority folder for synchronization

Synopsis:

```
#include <mme/mme.h>

int mme_setpriorityfolder( mme_hdl_t *hdl,
                           uint64_t folderid );
```

Arguments:

hdl An MME connection handle.

folderid The ID of the folder to be synchronized first. This ID must match the *folderid* field in the MME database **folders** table.

Library:

mme

Description:

The function *mme_setpriorityfolder()* tells the MME to synchronize the specified folder first. When you call this function, if the MME is in the process of sychronizing a mediastore, it pauses and synchronizes the specified folder first before resuming the rest of the sychronization.

Events

None delivered.

Blocking and validation

This function blocks on **qdb**. It validates:

- the folder ID (it must exist)
- that the synchronizer supports the use of prioritized folders

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), mme_resync_mediastore(), mme_sync_cancel(), mme_sync_directed(), mme_sync_file(), mme_sync_get_msid_status(), mme_sync_get_status()

mme_setrandom()

Sets the random playback mode for a control context

Synopsis:

```
#include <mme/mme.h>

int mme_setrandom( mme_hdl_t *hdl,
                   int mode );
```

Arguments:

hdl An MME connection handle.

mode The random mode. For a list of random modes, see **mmme_mode_random_t**.

Library:

mme

Description:

The function *mme_setrandom()* sets the random playback mode for a control context. Tracks are played in pseudo-random order (using the QDB *random()* function), and won't be repeated until all the tracks in the track session have already been played.

Clearing a track session

You can clear a track session by:

- calling *mme_stop()* to stop the track session
- calling *mme_settrksession()* with *trksessionid* set to 0 (zero)

A random or repeat mode setting only works if the external device supports the setting. If the external device doesn't support the requested setting, the MME logs a warning and continues playback.

CAUTION: A call to *mme_settrksession()* or *mme_set_msid_resume_trksession()* regenerates the pseudo-random list the MME uses for random mode playback.

Switching modes

The following describes how the MME plays through a track session when it switches between random and sequential modes, assuming that repeat mode is off.

When the MME switches the track session from sequential to random mode it:

- generates a list of all the tracks in the track session in pseudo-random order

- plays through this list until it has played all the tracks in the track session

When the MME switches the track session from random to sequential mode it:

- clears the random history
- continues playing tracks from the track session track list, starting with the currently playing track
- plays through the track session to the end. Tracks on the track session track list that are before the track at which sequential mode was started are not played.

If the client application calls *mme_setrandom()* when the track session is already in random mode, the MME clears all random history and:

- If the call to *mme_setrandom()* sets the playback mode to random mode (for example, from *random all* to *random album*, or from *random all* to *random all* [sic], the MME generates a new pseudo-random list of tracks in the track session and continues playback from this new list.
- If the call to *mme_setrandom()* turns off the random mode, the MME continues playback of the current track session in sequential mode.

For information about how the MME counts tracks played, see *mme_trksession_get_info()*.

CAUTION: The client application should always call the function *mme_trksession_get_info()* immediately after switching between random and sequential modes. Changing the random mode resets the value of *current_trk*, and if the client application doesn't update this information with *mme_trksession_get_info()* the client application can't know where it is in the track session.

Events

MME_EVENT_RANDOMCHANGE when the function has completed work.

Blocking and validation

This function blocks on control contexts. If *mme_setrandom()* is called and another function is called before *mme_setrandom()* returns, the second function blocks on **`io-media`** until *mme_setrandom()* returns. If there are no other pending calls, *mme_setrandom()* returns without blocking on **`io-media`**.

Returns:

≥**`0`** Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getrandom(), *mme_getrepeat()*, *mme_getscanmode()* *mme_setrepeat()*, **`mmme_mode_random_t`**, **`mmme_mode_repeat_t`**

mme_setrepeat()

Sets the repeat playback mode for a control context

Synopsis:

```
#include <mme/mme.h>

int mme_setrepeat( mme_hdl_t   *hdl,
                   int mode );
```

Arguments:

hdl An MME connection handle.

mode The repeat mode. For a list of repeat modes, see **mmme_mode_repeat_t**.

Library:

mme

Description:

The function *mme_setrepeat()* sets the repeat playback mode for a control context. If random playback mode is enabled and the repeat mode is MME_REPEAT_ALL, when all the tracks in a tracksession are played once, the MME determines a new pseudo-random order, and the first track in the new list starts playing. Playback will continue indefinitely.

If the repeat mode is MME_REPEAT_SINGLE, the current track repeats indefinitely.

A random or repeat mode setting only works if the external device supports the setting. If the external device doesn't support the requested setting, the MME logs a warning and continues playback.

Events

This function returns MME_EVENT_REPEATCHANGE when it has completed work.

Blocking and validation

This function blocks on control contexts. If *mme_setrepeat()* is called and another function is called before *mme_setrepeat()* returns, the second function blocks on **io-media** until *mme_setrepeat()* returns. If there are no other pending calls, *mme_setrepeat()* returns without blocking on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getrandom(), *mme_getrepeat()*, *mme_getscanmode()* *mme_setrandom()*, **`mmme_mode_random_t`**, **`mmme_mode_repeat_t`**

mme_setscanmode()

Set the scan mode and time for a control context

Synopsis:

```
#include <mme/mme.h>

int mme_setscanmode( mme_hdl_t *hdl,
                     uint64_t time );
```

Arguments:

hdl An MME connection handle.

time The number of milliseconds to play a track before skipping to the next track in the tracklist. Set to 0 to disable scan mode for the current control context.

Library:

mme

Description:

The function *mme_setscanmode()* sets the scan mode for a control context. The scan mode setting is the maximum number of milliseconds from the beginning of the track the MME will play before going to the next track.

If the scan mode setting is changed while a track is playing, the new scan mode will take effect immediately. The MME will behave as though the new setting had been made before it started playing the track. For example, if:

- the scan mode *time* is 8000 milliseconds
- the MME plays 5064 milliseconds of a track
- the scan mode *time* is set to 6000 milliseconds,

then the MME will stop playing the track at 6000 milliseconds and move to the next track.

If the scan mode *time* is set to a value less than the time already played from a track, the MME will move immediately to the next track.

Events

This function returns MME_EVENT_SCANMODECHANGE when it has completed work.

Blocking and validation

This function blocks on control contexts. If *mme_setscanmode()* is called and another function is called before *mme_setscanmode()* returns, the second function blocks on **io-media** until *mme_setscanmode()* returns. If there are no other pending calls, *mme_setscanmode()* returns without blocking on **io-media**.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_getrandom(), **<** *mme_getrepeat()*, *mme_getscanmode()*, *mme_setrandom()*, *mme_setrepeat()*

mme_settrksession()

Set the current track session

Synopsis:

```
#include <mme/mme.h>

int mme_settrksession( mme_hdl_t *hdl,
                       uint64_t trksessionid );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The track session ID, set by *mme_newtrksession()*; set to 0 to release ("unset") the current track session.

Library:

mme

Description:

The function *mme_settrksession()* sets the current track session for the specified control context. Before setting the track session for a control context, you must create the track session with *mme_newtrksession()*. To play the track session, after you have set it, call *mme_play()*.

A call to *mme_settrksession()* does the following:

- If the MME is playing a track and the file ID (*fid*) of this track is also in the newly requested track session, the MME does *not* stop playback. It :
  - seamlessly switches playback to the same track in the new tracksession
  - correctly sets the playback position for the new track session
  - if the newly set track session has more than one instance of the *fid* being played, playback is transferred to the first instance of this *fid*
- If the currently playing track is not in the newly requested track session, calling *mme_settrksession()* will stop the currently playing track session and clear the data associated with its track session. See "Preventing playback interruption" below.

- File-based track sessions are not permanent. Their contents are lost if playback is switched to another track session.
- Calling *mme_settrksession()* regenerates the list of tracks used by the MME for playback in random mode (the entries in the *randomid* field of the **`trksessionview`** table).

Preventing playback interruption

In order to not interrupt playback, *mme_settrksession()* will fail (return -1 and set *errno* to ECANCELED) if:

- the *fid* of the currently playing track is not in the new tracksession

or if:

- the track that was playing when the client issued the request to switch tracks is no longer playing

Client applications have several options for handling situations where *mme_settrksession()* cannot switch track sessions. These include:

- refuse the user request
- instruct the MME to stop playback, then set a new track session
- create a new track session that includes the *fid* for the currently playing track, then call *mme_settrksession()* again to attempt a seamless transition to the new track session

Using *mme_settrksession()* to resume playback

If you have stopped a track session and want to use *mme_trksession_resume_state()* to resume playback, you must call *mme_settrksession()* before calling *mme_trksession_resume_state()*, as follows:

1. Track session is stopped.
2. Call *mme_settrksession()*.
3. Call *mme_trksession_resume_state()*

For more information about stopping an resuming playback of track sessions, see "Stopping and resuming playback" in the chapter Playing Media in the *MME Developer's Guide*.

Releasing or "unsetting" a track session

You can release or "unset" the current track session by calling *mme_settrksession()* with *trksessionid* set to 0 (zero). Releasing a track session reduces the memory being used by the MME.

- You must call *mme_stop()* to stop the track session before you can release it.
- A track session can *not* be used by more than one control context. If you attempt to set a track session already in use by another control context, *mme_settrksession()* returns -1 and sets *errno* to EINVAL. To pass control of a track session to a new control context, you must first release it from the current control context

For information about deleting a track session, see “Deleting a track session” in the chapter Using the MME.

Events

If the tracksession being set is not the currently active track session, the MME delivers the event MME_EVENT_TRKSESSION. If the track session specified is already set, the MME delivers no events.

If the new track session has different repeat or random settings than the current settings on the control context, the MME delivers one or both of the events MME_EVENT_REPEATCHANGE and MME_EVENT_RANDOMCHANGE.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

This function blocks on control contexts. If *mme_settrksession()* is called and another function is called before *mme_settrksession()* returns, the second function blocks on **`io-media`** until *mme_settrksession()* returns. If there are no other pending calls, *mme_settrksession()* returns without blocking on **`io-media`**.

Returns:

`≥0` Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_newtrksession(), mme_rmtrksession(), mme_trksessionview_update()

Synopsis:

```
#include <mme/mme.h>

int mme_shutdown ( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_shutdown()* prepares the MME for shut down and delivers the event MME_EVENT_SHUTDOWN to all control contexts. When you call this function, it stops and disables:

- playback on all control contexts
- synchronizations on all control contexts
- any other MME operations that write to the MME database

After calling *mme_shutdown()*, you can:

1. Call *mme_disconnect()* to disconnect the client application from the MME.
2. Shut down the system by, for instance, turning off the power.

The function *mme_shutdown()* returns immediately and shuts down MME threads in the background. This behavior means that the MME may deliver other events *after* it has delivered MME_EVENT_SHUTDOWN. When all MME threads have shut down, the MME delivers the event MME_EVENT_SHUTDOWN_COMPLETED.

If you want to shut down the MME without turning off the system, after calling *mme_shutdown()* your client application needs to kill the MME process.

If your client application calls *mme_disconnect()* without calling *mme_shutdown()* first, it will disconnect from the MME control context, but the MME process will continue to run. Your client application will be able to use *mme_connect()* to make a new connection to the MME.

Events

This function returns the events MME_EVENT_SHUTDOWN and MME_EVENT_SHUTDOWN_COMPLETED.

Blocking and validation

Returns immediately and shuts down threads in background.

Returns:

≥`0` Success.

`-1` An error occurred (*errno* is set).

Examples:

The code snippet below illustrates how to shut down the MME.

```
mme_hdl_t *hdl = mme_connect("/dev/mme/default", 0);
mme_shutdown(hdl);
mme_disconnect(hdl);
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_connect(), *mme_disconnect()*

Slot type definitions

Synopsis:

```
#include <mme/interface.h>

#define MME_SLOTTYPE_*
```

Description:

The constants MME_SLOTTYPE_* define the slot types the MME recognizes. The values listed in the table below are used by the *slottype* field in the:

- **mme_play_info_t** data structure
- **slots** table

Constant	Value	Description
MME_SLOTTYPE_UNKNOWN	0	Unknown device.
MME_SLOTTYPE_USB	1	USB device.
MME_SLOTTYPE_CD	2	Internal CD/DVD drive.
MME_SLOTTYPE_DRIVE	3	Not used.
MME_SLOTTYPE_MEDIAFS	4	Not used.
MME_SLOTTYPE_CD_EXT	5	External CD/DVD drive.
MME_SLOTTYPE_CD_CHGR_INT	6	Internal CD/DVD changer.
MME_SLOTTYPE_CD_CHGR_EXT	7	External CD/DVD changer.
MME_SLOTTYPE_FILESYSTEM	8	Generic POSIX filesystem type.
MME_SLOTTYPE_BLUETOOTH	9	Bluetooth stack.
MME_SLOTTYPE_INTERNET	10	Internet, used for streaming.

Macros for determining slot types

The MME includes some macros that facilitate determining a slot type.

check_slottype_cd

Use the macro **check_slottype_cd** to determine if the slot type is for any type of CD:

```
#define check_slottype_cd(slottype) \
  ((slottype == MME_SLOTTYPE_CD || slottype == MME_SLOTTYPE_CD_EXT || \
    slottype == MME_SLOTTYPE_CD_CHGR_INT || slottype == MME_SLOTTYPE_CD_CHGR_EXT))
```

check_slottype_cd_int

Use the macro **check_slottype_cd_int** to determine if the slot type is for an internal CD:

```
#define check_slottype_cd_int(slottype) \
  ((slottype == MME_SLOTTYPE_CD || slottype == MME_SLOTTYPE_CD_CHGR_INT))
```

check_slottype_cd_int

Use the macro **check_slottype_cd_ext** to determine if the slot type is for an external CD:

```
#define check_slottype_cd_ext(slottype) \
  ((slottype == MME_SLOTTYPE_CD_EXT || slottype == MME_SLOTTYPE_CD_CHGR_EXT))
```

is_mediafs_type

Use the macro **is_mediafs_type** to determine if the slot type is for a media filesystem:

```
#define is_mediafs_type(SLOTTYPE) \
  ((SLOTTYPE == MME_SLOTTYPE_MEDIAFS) || (SLOTTYPE == MME_SLOTTYPE_MEDIAFS_2WIRE))
```

Classification:

QNX Multimedia

See also:

mme_play_info_t

mme_start_device_detection()

Start device and mediastore detection

Synopsis:

```
#include <mme/mme.h>

int mme_start_device_detection( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_start_device_detection()* starts device and mediastore detection. By default, device and mediastore detection is on, though it is possible to turn detection off when first starting the MME by changing the setting of **<DeviceDetection>** in the MME configuration file: **mme.conf**. For more information, see the chapter Configuring Device Support in the *MME Configuration Guide*.

CAUTION: If you have configured your MME to *not* automatically start device detection, always call *mme_start_device_detection()* before attempting any tasks that access devices (synchronization, playback, media copy and ripping, etc.).

Failure to call *mme_start_device_detection()* before attempting these type of tasks will produce unexpected results that may compromise the integrity of your system.

Events

None delivered.

Blocking and validation

Full validation of data; all arguments are checked before the call returns.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

Configuring Device Support in the *MME Configuration Guide*

Synopsis:

```
#include <mme/mme.h>

int mme_stop( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_stop()* stops the track session currently playing in the specified control context. You can call this function even if no track session is playing.

Events

MME_EVENT_PLAYSTATE with **mme_event_data_t.playstatespeed** set to 0 (zero).

Blocking and validation

This function verifies that the track session in the control context is in playback mode and can be stopped.

This function blocks on control contexts. If *mme_stop()* is called and another function is called before *mme_stop()* returns, the second function blocks on **io-media** until *mme_stop()* returns. If there are no other pending calls, *mme_stop()* returns without blocking on **io-media**.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_play()

Storage type definitions

Synopsis:

```
#include <mme/interface.h>

#define MME_STORAGETYPE_*
```

Description:

The constants MME_STORAGETYPE_* define the storage types the MME recognizes. The values listed in the tables below are used by the *storage_type* field in the:

- **`mme_play_info_t`** data structure
- **`mediastores`** table

Users can define their own, custom storager types, as required. Available values are listed in the table “User defined storage types” below.

Pre-defined storage types

Thes storage types are pre-defined:

Constant	Value	Description
MME_STORAGETYPE_UNKNOWN	0	Unknown storage type
MME_STORAGETYPE_AUDIOCD	1	Audio CD
MME_STORAGETYPE_FS	2	RAM disc
MME_STORAGETYPE_DEVB	2	MME_STORAGETYPE_FS
MME_STORAGETYPE_DVDAUDIO	3	Audio DVD
MME_STORAGETYPE_VCD	4	Video CD
MME_STORAGETYPE_SVCD	5	Super Video CD
MME_STORAGETYPE_DVDVIDEO	6	Video DVD
MME_STORAGETYPE_IPOD	8	iPod device
MME_STORAGETYPE_KODAKCD	9	Kodak picture CD
MME_STORAGETYPE_PICTURECD	10	Other picture CD
MME_STORAGETYPE_A2DP	12	A2DP protocol for Bluetooth
MME_STORAGETYPE_RESERVED0	13	Placeholder for UPnP
MME_STORAGETYPE_SMB	14	MME_STORAGETYPE_FS
MME_STORAGETYPE_FTP	15	Internet FTP connection

continued...

Constant	Value	Description
MME_STORAGETYPE_HTTP	16	Internet HTTP connection
MME_STORAGETYPE_NAVIGATION	17	Navigation CD or DVD. See also “Mediastore synchronization settings” .
MME_STORAGETYPE_UPGRADE	18	Upgrade CD or DVD. See also “Filtering synchronization by storage type”.
MME_STORAGETYPE_PLAYSFORSURE	20	PlaysForSure and similar devices.
MME_STORAGETYPE_UPNP	21	Devices using UPnP protocol.
MME_STORAGETYPE_INTERNETSTREAM	22	Internet streaming.

Multiple mediastore types on single device

These storage types are used to identify different mediastore types on the same device (such as a CD changer):

Constant	Value	Description
MME_STORAGETYPE_MEDIAFS_2WIRE_UNKNOWN	620	Unknown storage type
MME_STORAGETYPE_MEDIAFS_2WIRE_CDAUDIO	621	Audio CD
MME_STORAGETYPE_MEDIAFS_2WIRE_VCD	622	Video CD
MME_STORAGETYPE_MEDIAFS_2WIRE_DEVB	623	RAM disk
MME_STORAGETYPE_MEDIAFS_2WIRE_DVDAUDIO	624	Audio DVD
MME_STORAGETYPE_MEDIAFS_2WIRE_DVDVIDEO	625	Video DVD

User defined storage types

These storage types are available for custom implementations:

Constant	Value	Description
MME_STORAGETYPE_CUSTOM1	100	
MME_STORAGETYPE_CUSTOM2	101	
MME_STORAGETYPE_CUSTOM3	102	

continued...

Constant	Value	Description
MME_STORAGETYPE_CUSTOM4	103	
MME_STORAGETYPE_CUSTOM5	104	
MME_STORAGETYPE_CUSTOM6	105	
MME_STORAGETYPE_CUSTOM7	106	
MME_STORAGETYPE_CUSTOM8	107	
MME_STORAGETYPE_CUSTOM9	108	
MME_STORAGETYPE_CUSTOM10	110	

Events

Blocking and validation

Classification:

QNX Multimedia

See also:

MME_FORMAT_*, FTYPE_*, MME_MSCAP_*, MME_SYNC_OPTION_*, Table: **`mediastores`**

mme_sync_cancel()

Cancels mediastore synchronization

Synopsis:

```
#include <mme/mme.h>

int mme_sync_cancel ( mme_hdl_t *hdl,
                      uint64_t msid );
```

Arguments:

hdl An MME connection handle.

msid The ID for the mediastore on which synchronization is to be stopped or cancelled.

Library:

mme

Description:

The function *mme_sync_cancel()* cancels mediastore synchronizations. Set the parameter *msid* to the mediastore ID of the mediastore for which you want to cancel synchronization.

If you set the parameter *msid* to 0, *mme_sync_cancel()* cancels all current and pending mediastore synchronizations on all devices.

All cancelled synchronizations send an MM_EVENT_SYNCABORTED event.

For an active synchronization the MME:

- aborts the synchronization
- reports an error in the logs
- sends an MME_EVENT_SYNCABORTED event.

For pending synchronizations the MME

- immediately removes the pending synchronizations from the pending queue
- sends the MME_EVENT_SYNCABORTED event

Events

This function can return synchronization error events (MME_SYNC_ERROR_*) and MME_EVENT_SYNCABORTED.

Blocking and validation

This function is non-blocking. It delivers a MME_EVENT_SYNCABORTED event for each completed cancellation. It does not validate the mediastore ID (*msid*).

Returns:

≥0 Success: the mediastore synchronization was cancelled, or the mediastore was not being synchronized when the cancellation request was made.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), *mme_playlist_sync()*, *mme_resync_mediastore()*, *mme_setpriorityfolder()*, *mme_sync_directed()*, *mme_sync_file()*, *mme_sync_get_msid_status()*, *mme_sync_get_status()*

mme_sync_db_check()

Check and repair a folder with inconsistencies

Synopsis:

```
#include <mme/mme.h>

int mme_sync_db_check( mme_hdl_t *hdl,
                       uint64_t folderid,
                       uint32_t flags );
```

Arguments:

hdl	An MME connection handle.
folderid	The ID of the folder to verify and repair.
flags	Flags controlling the verification and repair. See “Flags” below.

Library:

mme

Description:

The function *mme_sync_db_check()* checks the specified folder for consistency and, optionally, attempts to repair any errors it encounters. It:

- Checks folder information (in fields reserved for internal use only) in the **folders** table.
- Logs all inconsistencies.
- If the MME_SYNC_OPTION_REPAIR flag is set, attempts to repair any inconsistencies that it finds between the folder information in the database and the folder’s contents.

The function *mme_sync_db_check()*:

- can be used to try to repair inconsistencies:
  - if problems are encountered after a synchronization
  - for mediastores with POSIX compliant filesystems *only*; if the specified folder is not on a mediastore with a POSIX compliant filesystem (i.e. a CDDA), *mme_sync_db_check()* returns an error
- always verifies the consistency of the specified folder if it can

CAUTION: If *mme_sync_db_check()* finds and is unable to repair inconsistencies between the MME database and a folder, there is probably a problem with the database that requires immediate attention.

When and how to use *mme_sync_db_check()*

You should use *mme_sync_db_check()* if you suspect a problem with the MME database, and proceed as follows:

1 Call *mme_sync_db_check()* to verify the folder that may be the source of the problem (do *not* set *flags* to MME_SYNC_OPTION_REPAIR). If the function reports zero inconsistencies, the database does not require repair.

2 If *mme_sync_db_check()* reports and logs inconsistencies, call the function again with the *flags* option set to MME_SYNC_OPTION_REPAIR.

3 After *mme_sync_db_check()* finishes repairing the database, run this function again, with the *flags* option *not* set to MME_SYNC_OPTION_REPAIR — you need to verify that the repair was completely successful.

4 If *mme_sync_db_check()* still reports inconsistencies:

4a Contact QNX and forward, if possible:

- all logs
- the database with the inconsistencies
- a copy of the mediastore associated with the inconsistencies; this copy must keep all file modification times from the original

4b Restart and resynchronize the mediastore with the inconsistencies by calling *mme_ms_restart()* to delete all database contents associated with this mediastore.

4c If resynchronization of an newly active mediastore is not automatic on your system, call *mme_resync_mediastore()* to synchronize the mediastore.

5 If *mme_sync_db_check()* no longer reports inconsistencies, resynchronize the mediastore by calling *mme_resync_mediastore()*.

Flags

The behavior of *mme_sync_db_check()* is determined by the values of the *flags* argument:

- MME_SYNC_OPTION_REPAIR (0x0400) — verify and attempt to repair the database
- MME_SYNC_OPTION_VERIFY (0x0800) — verification is the minimum action performed by *mme_sync_db_check()*, so this flag is always implied in any call to this function

- MME_SYNC_OPTION_RECURSIVE (0x4000) — verify and repair recursively (the specified folder and its subfolders)

Events

None delivered.

Blocking and validation

This function checks that:

- the specified folder ID exists
- the specified folder is on an active mediastore
- there is a checking function

This function runs synchronously, and therefore blocks.

Returns:

0 Success: the verification or repair operation has started.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

MME_SYNC_OPTION_*

Synopsis:

```
#include <mme/mme.h>

int mme_sync_directed( mme_hdl_t *hdl,
                       uint64_t msid,
                       const char *path,
                       uint32_t options );
```

Arguments:

hdl An MME connection handle.

msid The ID for the mediastore on which directed synchronization is to be performed.

path The path to be synchronized on the mediastore.

options The synchronization options. The options can be any combination of:

- MME_SYNC_OPTION_CANCEL_CURRENT — Cancel any other synchronization on the mediastore, and run this directed synchronization. Used only by *mme_sync_directed()*; not used by *mme_resync_mediastore()*.
- MME_SYNC_OPTION_CLR_INV_COPIED — set to 0 (zero) all invalid *copied_fid* values in the **library** table.
- MME_SYNC_OPTION_PASS_FILES — synchronize files (perform first pass synchronization).
- MME_SYNC_OPTION_PASS_METADATA — synchronize metadata (perform second pass synchronization).
- MME_SYNC_OPTION_PASS_PLAYLISTS — synchronize playlists (perform third pass synchronization).
- MME_SYNC_OPTION_PASS_ALL — synchronize files, metadata, and playlists.
- MME_SYNC_OPTION_RECURSIVE — perform a recursive synchronization starting from the path defined by *path*. Assumed set by *mme_resync_mediastore()*.

Library:

mme

Description:

The function *mme_sync_directed()* starts directed synchronization for a specified path on a mediastore.

Directed synchronization allows you to synchronize only a specified path on a mediastore. This capability is particularly useful if you want to synchronize part of a large mediastore in order to start playing its contents, then synchronize the rest (or other parts) of the mediastore in the background or at a later time.

Directed synchronization is only available for mediastores with hierarchical directory structures: HHDs, iPods, USB sticks, data CDs, etc. It is not available for mediastores, such as music CDs, that have a single-level directory structure.

CAUTION: A clean up of invalid *copied_id* fields can take a long time. Use the MME_SYNC_OPTION_CLR_INV_COPIED flag judiciously — *only* when synchronizing after deleting media files from your database.

Events

This function returns synchronization events with the operation ID. See the chapter MME Synchronization Events for a full list.

Blocking and validation

This function is non-blocking. It returns asynchronously. On completion, it returns a positive integer, which is the operation ID. This return value is sent with the event:

- MME_EVENT_MS_SYNCCOMPLETE if the operation was successfully completed
- MME_EVENT_SYNCABORTED if the operation failed to complete successfully

Returns:

`>0` Success: the operation ID of the directed synchronization.

`-1` An error occurred (*errno* is set).

Examples:

The code snippet below shows an example of how directed synchronization can be used. It is taken from the sample application **`mmebrowse`** that browses both synchronized and unsynchronized mediastores.

```
uint64_t go_to_folder (
    mme_hdl_t *mme,
    qdb_hdl_t *db,
    uint64_t msid,
    uint64_t folderid,
    const char *folder_name
```

```
)
{
    int rc;

    /* if it's already synced, don't resync it unless forced */
    if ( force_resync || ( ! folder_synced( db, msid, folderid ) ) ) {
        rc = mme_sync_directed( mme, msid, folder_name, MME_SYNC_OPTION_PASS_ALL );
        if ( rc == -1 ) {
            fprintf( stderr, "Unable to get sync path \"%s\": %s (%d).\n",
                     folder_name, strerror( errno ), errno );
            return 0;
        }

        if ( waitfor_directed_syncevent( rc ) != 1 ) {
            /* operation didn't finish, or failed */
            fprintf( stderr, "**** Operation failed. ****\n" );
            return 0;
        }
    }
    return folderid;
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), *mme_playlist_sync()*, *mme_resync_mediastore()*, *mme_setpriorityfolder()*, *mme_sync_cancel()*, *mme_sync_file()*, *mme_sync_get_msid_status()*, *mme_sync_get_status()*

mme_sync_file()

Synchronize a specified file.

Synopsis:

```
#include <mme/mme.h>

int mme_sync_file( mme_hdl_t *hdl,
                   uint64_t old_fid,
                   uint64_t new_msid,
                   const char *new_filename );
```

Arguments:

hdl	An MME connection handle.
old_fid	The file ID of the file in the library before the change. Use 0 for file additions, to indicate that there is no existing file associated with the operation.
new_msid	The ID for the media store with the *new_filename* path for the new file. This value may be 0 if *new_filename* is NULL, as in the case of file removals.
new_filename	The path and name of the new file, relative to the mountpath of the mediastore identified by *new_msid*. This value may be a NULL pointer to indicate there is no new file associated with the operation, as in the case of file removals.

Library:

mme

Description:

The function *mme_sync_file()* starts a synchronization for a specified file.

File synchronization allows the client application to have the MME synchronize only a specified file. This capability is typically used when the client application knows that a specific file change has occured: a file has been deleted, added, moved or renamed.

In all cases, the client application must specify, as a minimum, one of the *old_fid* or the *new_filename*. The values the client application should assign to these variables before passing them to *mme_sync_file()* depending on the reason it is calling the function:

- File additions

old_fid	0.
new_filename	The path and name of the new file.

- File changes

old_fid	fid of the changed file.
new_filename	The path and name of the changed file.

- File removals

old_fid	The file ID (*fid*) of the deleted file.
new_filename	NULL.

Function behavior

No synchronization options are available for this function; it attempts to do the equivalent of both file and metadata synchronization passes.

File changes and additions

During synchronization, the *mme_sync_file()* delivers synchronization events:

- When the function begins synchronization, it delivers the event MME_EVENT_MS_SYNC_STARTED with the operation ID and the *msid* of the new file.
- If *old_fid* is not specified and the file exists, the function delivers the event MME_EVENT_MS_SYNCFIRSTFID with the *fid* of the file in the MME database. The function performs the first and second synchronization passes, but delivers only the event MME_EVENT_MS_1PASSCOMPLETE.
- If *old_fid* is specified, the function updates the existing library with the new folder ID, mediastore ID and filename, but makes no other changes to the metadata. Before completion it delivers only the event MME_EVENT_MS_1PASSCOMPLETE.

File removal

If *new_msid* is **`0`** and *new_filename* is NULL, *mme_sync_file()* removes the file specified by *fid*. The function returns 0 on successful completion.

Limitations

The function *mme_sync_file()* can only be used with certain media store types. For example, the function it is not supported for use with iPods.

There is no support for changes across mediastores. For example, when both the *msid* and *old_msid* are specified, the *msid* for the old file must match the *old_msid*.

File move or rename is supported only when the file remains on the same media store. In this case, all metadata about the file is preserved. If the file is moved to a different mediastore, two separate calls to *mme_sync_file()* are required and:

- the file ID of the renamed file may change
- metadata is not preserved

Events

This function returns synchronization events with the operation ID. See “File changes and additions” above, and the chapter MME Synchronization Events for a full list.

Blocking and validation

This function is non-blocking. It returns synchronously. On completion, it returns 0 or a positive integer, which is the operation ID. This return value is sent with:

- an MME_EVENT_MS_SYNCCOMPLETE event if the operation was successfully completed
- an MME_EVENT_SYNCABORTED event if the operation failed to complete successfully

Returns:

≥0 Success:

- **=0** Operation completed synchronously. This situation occurs only if *new_msid* is **0** and *new_filename* is NULL.
- **>0** Value returned is synchronization operation ID. the operation ID of the directed synchronization.

-1 An error occurred (*errno* is set). The event MME_EVENT_SYNCABORTED is sent with the *msid* and the operation ID.

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), *mme_resync_mediastore()*, *mme_setpriorityfolder()*, *mme_sync_cancel()*, *mme_sync_directed()*, *mme_sync_get_msid_status()*, *mme_sync_get_status()*

mme_sync_get_msid_status()

Gets the status of synchronization for a mediastore

Synopsis:

```
#include <mme/mme.h>

int mme_sync_get_msid_status ( mme_hdl_t *hdl,
                               uint64_t msid,
                               mme_sync_status_t *status )
```

Arguments:

hdl	An MME connection handle
msid	The ID of the mediastore for which you want to get the synchronization status.
status	A pointer to a **mme_sync_status_t** structure where the function can store information about the synchronization status.

Library:

mme

Description:

The function *mme_sync_get_msid_status()* gets information about a specific mediastore's synchronization status. For more information about the *status* structure, see **mme_sync_status_t**.

If you request the synchronization status for an invalid MSID (a mediastore that doesn't exist), the function returns success, but all pass fields in *status* are filled with 0.

Events

None delivered.

Blocking and validation

This function is non-blocking. It validates that *status* is not null.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_directed_sync_cancel(), mme_resync_mediastore(), mme_setpriorityfolder(), mme_sync_cancel(), mme_sync_directed(), mme_sync_file(), mme_sync_get_msid_status()

mme_sync_get_status()

Gets information about system synchronization

Synopsis:

```
#include <mme/mme.h>

int mme_sync_get_status ( mme_hdl_t *hdl,
                          mme_sync_status_t *status,
                          size_t status_size )
```

Arguments:

hdl	An MME connection handle.
status	A pointer to an array of **mme_sync_status_t** structures where the function can store status information. Pass as NULL to get the number of mediastores actively involved in synchronization. See **mme_sync_status_t** in this reference.
status_size	The number of elements in the *status* array. It may be 0 (zero).

Library:

mme

Description:

The function *mme_sync_get_status()* gets information about system synchronization. You can call this function and pass *status* as NULL and *status_size* as 0 to simply return the number of mediastores that have synchronization passes underway or pending, and use this information to set up the *status* array for a subsequent call. However, keep in mind that mediastore synchronization status can change rapidly, so you should always check the return value for the number of elements that contain valid data in *status*.

Events

None delivered.

Blocking and validation

This function is non-blocking.

Returns:

≥**0**	Success. The value returned is the number of media stores that have synchronization passes in progress or pending.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_playlist_sync(), mme_sync_cancel(), mme_sync_get_msid_status()
mme_sync_status_t()

Synchroniztion option type definitions

Synopsis:

```
#include <mme/interface.h>

#define MME_SYNC_OPTION_*
```

Description:

The MME_SYNC_OPTION_* constants are bit masks defining the synchronization options that can be set for synchronizing mediastores. The values listed in the table below can be set by the client application to configure synchronization behaviors.

For more information, see the chapter Synchronizing Media, and *mme_sync_directed()*, *mme_resync_mediastore()* and *mme_sync_get_status()* in this reference.

Constant	Value	Description
MME_SYNC_OPTION_PASS_FILES	0x0001	Perform file and folder synchronization pass.
MME_SYNC_OPTION_PASS_METADATA	0x0002	Perform metadata synchronization pass.
MME_SYNC_OPTION_PASS_PLAYLISTS	0x0004	Perform playlist synchronization pass.
MME_SYNC_OPTION_PASS_EXT_DB_SYNC	0x0008	Perform external database synchronization pass.
MME_SYNC_OPTION_PASS_ALL	0x000F	Perform all synchronization passes: FILES + METADATA + PLAYLISTS + EXT_DB_SYNC.
MME_SYNC_OPTION_REPAIR	0x0400	Repair the database. See *mme_sync_db_check()*.
MME_SYNC_OPTION_VERIFY	0x0800	Verify if the database needs repairing. See *mme_sync_db_check()*.
MME_SYNC_OPTION_CLR_INV_COPIED	0x1000	Set to 0 (zero) all invalid *copied_fid* values in the **`library`** table. This option can be used only with *mme_sync_directed()* or *mme_resync_mediastore()*.

continued...

Constant	Value	Description
MME_SYNC_OPTION_CANCEL_CURRENT	0x2000	Cancel current synchronization.
MME_SYNC_OPTION_RECURSIVE	0x4000	Perform recursive synchronization.
MME_SYNC_OPTION_BLOCKING	0x8000	For future use.

Classification:

QNX Multimedia

See also:

MME_FORMAT_*, MME_FTYPE_*, MME_MSCAP_*, MME_STORAGETYPE_*, *mme_sync_db_check()*, *mme_sync_directed()*, *mme_resync_mediastore()*, *mme_sync_get_status()*

mme_sync_set_debug()

Set the verbosity level for synchronization operations

Synopsis:

```
#include <mme/mme.h>

int mme_sync_set_debug( mme_hdl_t *hdl,
                        uint8_t verbose,
                        uint8_t debug );
```

Arguments:

hdl	An MME connection handle.
verbose	The new verbosity setting.
debug	For future use.

Library:

mme

Description:

The function *mme_sync_set_debug()* sets the verbosity level for synchronization operations. It does not affect the verbosity levels of other operations.

You can call *mme_sync_set_debug()* at any time to change the amount of information synchronizations write to the log files. You can use it to help you debug and tune your implementation. For example, you can call it before a second synchronization pass to increase the information logged during this pass, then call it again when the pass is complete to lower the verbosity for other synchronization operations.

Events

None delivered.

Blocking and validation

This function makes no validations and does not block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_set_api_timeout(), mme_set_debug()

mme_sync_status_t

Synchronization status information

Synopsis:

```
#include <mme/types.h>

typedef struct s_mme_sync_status {
    uint64_t    msid;
    uint16_t    passes_done;
    uint16_t    current_pass;
    uint16_t    passes_to_do;
    uint16_t    reserved[1];
    uint32_t    operation_id;
} mme_sync_status_t;
```

Description:

The structure **mme_sync_status_t** carries information about the status of a synchronization operation. It has at least the members described in the table below.

Member	Type	Description
msid	**uint64_t**	The ID of the mediastore.
passes_done	**uint16_t**	The synchronization passes that have completed.
current_pass	**uint16_t**	The current synchronization pass flag.
passes_to_do	**uint16_t**	The synchronization passes yet to be performed.
operation_id	**uint32_t**	An identifier for the synchronization operation, used for directed synchronizations. The MME sets it to 0 (zero) for all synchronizations, *except* directed synchronizations.

Pass flags

The *passes_done* and *passes_to_do* are a combination of zero or more of the flags with the values listed below:

- MME_SYNC_OPTION_PASS_FILES — file pass
- MME_SYNC_OPTION_PASS_METADATA — metadata pass
- MME_SYNC_OPTION_PASS_PLAYLISTS — third pass
- MME_SYNC_OPTION_PASS_ALL — all passes

The *current_pass* flag can only be set to 0 (zero) or 1 (one).

The MME_SYNC_OPTION_PASS_* constants are described in MME_SYNC_OPTION_* in this reference.

Classification:

QNX Multimedia

See also:

mme_sync_get_status()

Synopsis:

```
#include <mme/types.h>

typedef struct _mme_time_info {
    uint64_t    time;
    uint64_t    duration;
} mme_time_t;
```

Description:

The structure **mme_time_t** carries the total play time and the play time elapsed for the current track or file. It is used during operations such as playback and ripping. It includes at least the members listed in the table below.

Member	Type	Description
time	**uint64_t**	The current time position in the track or file, in milliseconds.
duration	**uint64_t**	The total duration of the track or file, in milliseconds.

Classification:

QNX Multimedia

See also:

mme_play_get_status()

mme_trksession_append_files()

Append tracks or a stream to a file-based track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_append_files( mme_hdl_t *hdl,
                                 uint64_t trksessionid,
                                 int nfiles,
                                 uint64_t *msid,
                                 const char **filename );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The ID of the track session to update.
nfiles	The number of files to append to the track session.
msid	A pointer to an array of mediastore IDs matching the *filename* array. Each *msid* in the *msid* array must: • match the mediastore FTYPE_DEVICE file ID *fid* that was used to create the track session • identify the mediastore with the file at the same location in the *filename* array; for example, index 12 of the *msid* array referes to the mediastore ID of the filename at index 12 of the *filename* array.
filename	A pointer to an array of filenames of tracks to be played, or the URL of a stream to be played. If *filename* points to an array of tracks, it includes includes the path to the file on the mediastore, but it does *not* include the mountpath to the mediastore. The path in *filename* must begin with a "/" (slash). For example: **/songs_folder/album_folder/**.

Library:

mme

Description:

The function *mme_trksession_append_files()* appends files (or streams) to an existing file-based track session. It can be used to add to a track session tracks of interest discovered through the explorer API, subject to the following conditions:

- The file or files to be appended are on the same mediastore (the same FTYPE_DEVICE) that was used to create the track session.
- The track session is not in repeat or random mode.

When *mme_trksession_append_files()* successfully appends a file, files or a stream to a track session it delivers an MME_EVENT_TRKSESSIONVIEW_UPDATE event to indicate to the client application that the track session has changed.

Events

MME_EVENT_TRKSESSIONVIEW_UPDATE.

Blocking and validation

This function doesn't block.

Returns:

≥`0`	Success.
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_clear_files(), *mme_trksession_get_info()*, *mme_trksession_resume_state()*, *mme_trksession_save_state()*, *mme_trksession_set_files()*

mme_trksession_clear_files()

Clear all files from a file-based track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_clear_files( mme_hdl_t *hdl,
                                uint64_t trksessionid );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The ID of the track session to clear.

Library:

mme

Description:

The function *mme_trksession_clear_files()* clears all tracks from the specified *file-based* track session. You must stop playback before calling this fucntion.

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), mme_trksession_get_info(), mme_trksession_resume_state(), mme_trksession_save_state(), mme_trksession_set_files()

mme_trksession_get_info()

Get information about the current track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_get_info( mme_hdl_t *hdl,
                             uint64_t *trksessionid,
                             uint64_t *current_trk,
                             uint64_t *total_trk );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The ID of the current track session, unique for the control context.
current_trk	The one-based track currently being played from the tracksession. For more information, see "Track number count with sequential and random modes" below.
total_trk	The number of tracks in the current track session.

Library:

mme

Description:

The function *mme_trksession_get_info()* retrieves the following information about the current track session:

- the track session ID
- the *fid* of the track currently being played
- the total number of tracks in the track session.

This information provides a snapshot of a track session and what the MME is doing with the track session. For example, a track session with *total_trk* set to 0 (zero) indicates that the MME found no tracks or files that meet the criteria used to create the track session (artist, genre, etc.).

Always use this function to retrieve track session information. The MME may to need to retrieve track session information from an external device, such as an iPod, because information stored in an external device will not be available in the **trksession** table.

Don't use the track session table **trksession** to retrieve track session information, because this method will miss information on external devices.

The values of a track's *sequentialid* and *randomid* fields in the **`trksessionview`** table have no bearing on the value of *current_trk*. The value returned in *current_trk* is just the one-based offset in the track session of the currently playing track. For example, in a track session with 10 tracks, if playback is at the third track, *current_trk* will be 3, while the *sequentialid* field for the track may be 7, or some other number used to sort the tracks (ORDER BY) when the track session was created.

Track number count with sequential and random modes

The method used by the MME to count the tracks played in a track session differs in sequential and random modes, and is consistent with the method used by iPods.

Sequential mode

For track sessions in sequential mode, the MME assigns *current_trk* the number of the track in the track session, and increments its value by 1 (one) each time it begins playing a new track. For example, if the end-user chooses to start playing in sequential mode on track 3 of the track session, *current_track* the value of *current_track* will be 3. The MME will continue playing tracks to the end of the track session, but will not play tracks 0, 1 or 2 (unless repeat mode is on, in which case the MME will continue playing through the track list until instructed to stop). The value of *current_trk* is therefore always the same as the track number in the tracklist.

Random mode

When the MME is asked to start playing a track session in random mode, it uses the QDB *random()* function to create a pseudo-random order, and makes a list of tracks to play in this order. The MME assigns *current_trk* the value 0 (zero) when it starts playing the first track in its pseudo-random list,and increments this value by 1 (one) each time it begins playing a new track. Thus, the value of *current_trk* is the number or tracks played plus 1 for the current track, and has no relationship to the track number in the track session.

How to calculate the number of tracks left to play

For both sequential and random modes, to calculate the number of tracks left to play in the track session, simply subtract *current_trk* from *total_trk*. The end of the track session is reached when *current_trk=total_trk*.

Events

None delivered.

Blocking and validation

This function returns immediately.

Returns:

`>=0`	Success: the MME retrieved the track session information for the current track session..
`-1`	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_resume_msid(), *mme_set_msid_resume_trksession()*, *mme_trksession_resume_state()*, *mme_trksession_save_state()*, *mme_trksessionview_update()*

mme_trksession_resume_state()

Resume playing a track session at the last saved position

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_resume_state( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_trksession_resume_state()* resumes playing a track session at the point its state was saved by a call to *mme_trksession_save_state()*.

Before stopping a track session, you must use the function *mme_trksession_save_state()* to save its state. After you have saved the track session's state, you can stop playback, then call the functions *mme_settrksession()* and *mme_trksession_resume_state()* at any time to resume playback.

For more information about stopping and resuming playback of track sessions, see "Stopping and resuming playback" in the chapter Playing Media.

Events

This function may deliver any event of the class MME_EVENT_CLASS_PLAY, and any MME_PLAY_ERROR_* event.

Blocking and validation

This function does not verify that the *fid* is in the track session. If the connection to the MME is synchronous, the function validates that the file exists and that it is playable.

This function blocks on control contexts. If *mme_trksession_resume_state()* is called and another function is called before *mme_trksession_resume_state()* returns, the second function blocks on **io-media** until *mme_trksession_resume_state()* returns. If there are no other pending calls, *mme_trksession_resume_state()* returns without blocking on **io-media**.

Returns:

≥**0** Success: MME resumed playback of the track session.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), *mme_trksession_clear_files()*, *mme_trksession_get_info()*, *mme_trksession_save_state()*, *mme_trksession_set_files()*

mme_trksession_save_state()

Save the playing position of the current track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_save_state( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_trksession_save_state()* saves the playing position of the current track session. If you want to be able to stop a track session then resume playing it at a later time, you must use this function to save its playing position before you interrupt it.

If the track session is on a device, such as an iPod, that manages its own track sessions, do not call *mme_trksession_save_state()*. The device is responsible for saving its state, and will resume playback from the correct point when you call *mme_play_resume_msid()*.

For more information about stopping and resuming playback of track sessions, see "Stopping and resuming playback" in the chapter Playing Media.

Events

None delivered.

Blocking and validation

This function may block on the control context, **qdb** or **io-media**. Depending on the MME connection, it behaves as follows:

- Asynchronous — returns before saving the track session.
- Synchronous — validates that a track session is set, and returns only after the MME has saved the track session state or if it encounters a failure.

Returns:

≥0 Success: the MME saved the state of the current track session.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), mme_trksession_clear_files(), mme_trksession_get_info(), mme_trksession_resume_state(), mme_trksession_set_files()

mme_trksession_set_files()

Update the track list in a file-based track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksession_set_files( mme_hdl_t *hdl,
                              uint64_t trksessionid,
                              int nfiles,
                              uint64_t *msid,
                              const char **filename,
                              unsigned offset, );
```

Arguments:

hdl	An MME connection handle.
trksessionid	The ID of the track session to update.
nfiles	The number of files to set in the track session.
msid	A pointer to an array of mediastore IDs matching the *filename* array. Each *msid* in the *msid* array must: • match the mediastore FTYPE_DEVICE file ID *fid* that was used to create the track session • identify the mediastore with the file at the same location in the *filename* array; for example, index 12 of the *msid* array referes to the mediastore ID of the filename at index 12 of the *filename* array.
filename	A pointer to an array of filenames of tracks to be played. The filename includes the path to the file on the mediastore, but it does *not* include the mountpath to the mediastore. The path in *filename* must begin with a "/" (slash). For example: **`/songs_folder/album_folder/`**.
offset	The offset to jump to in the new track session.

Library:

`mme`

Description:

The function *mme_trksession_set_files()* replaces list of tracks to play in a *file-based* track session with a new list.

If the *offset* argument is not 0, this value is considered the the offset (position) in the new track session that the MME should go to when it begins playback. This offset in the new tracksession must match currently playing track.

When *mme_trksession_append_files()* successfully appends a file or files to a track session it delivers an MME_EVENT_TRKSESSIONVIEW_UPDATE event to indicate to the client application that the track session has changed.

Events

MME_EVENT_TRKSESSIONVIEW_UPDATE.

Blocking and validation

This function doesn't block.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), *mme_trksession_clear_files()*, *mme_trksession_get_info()*, *mme_trksession_resume_state()*, *mme_trksession_save_state()*

mme_trksessionview_get_current()

Get information about the current track

Synopsis:

```
#include <mme/mme.h>

int mme_trksessionview_get_current( mme_hdl_t *hdl,
                                    mme_trksessionview_info_t *info );
```

Arguments:

hdl The handle of the control context.

info A pointer to the information about the MME track session, the device track session, and (where applicable) the video chapters.

Library:

mme

Description:

The function *mme_trksessionview_get_current()* gets information about the current track and places it in the structure **mme_trksessionview_info_t**. It works exactly like *mme_trksession_get_info()* except that it assumes the current title (if applicable) and track.

Events

None delivered.

Blocking and validation

This function performs no validations and doesn't block.

Returns

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_trksession_get_info(), **mme_trksessionview_info_t**

mme_trksessionview_get_info()

Get information about a track in a track session

Synopsis:

```
#include <mme/mme.h>

int mme_trksessionview_get_info( mme_hdl_t *hdl,
                                 unsigned track,
                                 unsigned title,
                                 mme_trksessionview_info_t *info );
```

Arguments:

hdl The handle of the control context.

track The number of the track for which you want information, counted sequentially from track 0 in the track session. (The track number is zero-based.)

title The title on the device or DVD for which you want information, counted sequentially from title 1 in the device track session. (The title number is one-based.)

info A pointer to the information about the MME track session, the device track session, and (where applicable) the video chapters.

Library:

mme

Description:

The function *mme_trksessionview_get_info()* retrieves information about a title or track on a DVD or a device, such as an iPod, that manages its own track sessions, and places it in the structure **mme_trksessionview_info_t**.

Events

None delivered.

Blocking and validation

This function performs no validations and doesn't block.

Returns:

>0 Success

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksession_append_files(), *mme_trksession_clear_files()*, *mme_trksession_resume_state()*, *mme_trksession_save_state()*, *mme_trksession_set_files()*, *mme_trksession_get_current()*, **`mme_trksessionview_info_t`**

mme_trksessionview_info_t

Information about items found during mediastore exploration

Synopsis:

```
#include <mme/mme.h>

typedef struct {
    uint64_t        trksessionid;
    uint64_t        flags;
    uint32_t        track;
    uint32_t        totaltracks;
    uint32_t        title;
    uint32_t        ntitles;
    uint32_t        chapter;
    uint32_t        nchapters;
    char            reserved[16];
} mme_trksessionview_info_t;
```

Description:

The structure **mme_trksessionview_info_t** carries information about:

- tracks in the current MME track session
- titles (tracks) in the current device track session, for devices, such as iPods, that manage their own track sessions; and titles on DVDs
- chapters, in videos on DVDs and other mediastores and devices

It contains at least the members described in the table below.

Member	Type	Description
trksessionid	**uint64_t**	The ID of the current MME track session.
flags	**uint64_t**	For future use.
track	**uint32_t**	The ID of the current track in the MME track session.
totaltracks	**uint32_t**	The total number of tracks in the current MME track session.
title	**uint32_t**	The ID of the current title (track) in the device's track session.
ntitles	**uint32_t**	The total number of titles (tracks) in the current device track session.
chapter	**uint32_t**	For future use. The ID of the current chapter in the current title in the current track session on a DVD, or other mediastore or device.

continued...

Member	Type	Description
nchapters	`uint32_t`	For future use. The total number of chapters in the current title in the current track session on a DVD, or other mediastore or device.
reserved	`uint32_t`	For future use.

Classification:

QNX Multimedia

See also:

mme_trksession_get_current(), *mme_trksessionview_get_info()*, *mme_trksessionview_metadata_get()*, *mme_trksessionview_readx()*, *mme_trksessionview_writedb()*

mme_trksessionview_metadata_get()

Get metadata for a track in a track session

Synopsis:

```
#include <mme/mme.h>

mme_metadata_hdl_t *mme_trksessionview_metadata_get( mme_hdl_t *hdl,
                                                     unsigned track,
                                                     unsigned title,
                                                     unsigned chapter,
                                                     const char *types,
                                                     uint32_t flags );
```

Arguments:

hdl	The handle of the control context.
track	The number of the track for which you want information, counted sequentially from track 0 in the track session. (The track number is zero-based.)
title	The ID of the current title (track) in the device's track session.
chapter	For future use. The ID of the current chapter in the current title in the current track session on a DVD, or other mediastore or device.
types	The types of metadata requested. See the chapter Metadata and Artwork in the *MME Developer's Guide*.
flags	For future use.

Library:

mme

Description:

The function *mme_trksessionview_metadata_get()* retrieves metadata for a title or track on a DVD or a device, such as an iPod, that manages its own track sessions. It returns this metadata in the metadata structure **mme_metadata_hdl_t**.

Events

None delivered.

Blocking and validation

This function performs no validations, and doesn't block.

Returns:

≥**0**: data in **mme_metadata_hdl_t**.

	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksessionview_get_info(), **mme_trksessionview_info_t**, *mme_trksessionview_readx()*, *mme_trksessionview_writedb()*

mme_trksessionview_readx()

Read track data from the track session view

Synopsis:

```
#include <mme/mme.h>

int mme_trksessionview_readx( mme_hdl_t *hdl,
                              unsigned type,
                              int offset,
                              unsigned ntracks,
                              void *buf,
                              unsigned *buflen );
```

Arguments:

hdl	An MME connection handle.
type	The type of information requested. See TRKVIEW_READ_* below.
offset	The 0-based offset in the track session at which to start reading.
ntracks	The number of tracks for which information is requested.
buf	A pointer to the buffer into which the information can be placed.
buflen	A pointer to the size,in bytes, of the buffer. Specify the size you allocate for the request when calling the function; it will fill in the size actually required to fulfill the request, which you can check when the function returns.

Library:

mme

Description:

The function *mme_trksessionview_readx()* reads track session data from the **trksessionview** table. It fills the buffer referenced by *buf* with an array of elements.

The number of elements in the array is set by the *ntracks* parameter, and the type and size of the elements are determined by the *type* parameter:

- If you set *type* to TRKVIEW_READ_FID, *mme_trksessionview_readx()* fills the array with **trksessionview_entry_t** structures.
- If you set *type* to TRKVIEW_READ_FILE, *mme_trksessionview_readx()* fills the array with **trksessionview_entry_file_t** structures.

Set the *type* argument to TRKVIEW_READ_FILE only for file-based track sessions (track sessions created with the *mode* argument set to MME_PLAYMODE_FILE).

The function *mme_trksessionview_readx()* returns the number of elements it successfully read. This number may be less than the number of elements requested (*ntracks* if the source mediastore contains less files than the requested number, or if the allocated buffer is too small to contain the information for all the requested tracks.

To ensure that you call *mme_trksessionview_readx()* with a buffer large enough for all the requested elements, you can call it once with *buflen* set to 0:

```
*buflen = 0
mme_trksessionview_readx(hdl, type, offset, ntracks, buf, buflen);
```

The function will fill in *buflen* with the buffer size required for the number and type of information you request. You can then call *mme_trksessionview_readx()* a second time, certain that your buffer is large enough for your request.

trksessionview_entry_t

```
typedef struct {
        uint64_t fid;
} trksessionview_entry_t;
```

The data structure **trksessionview_entry_t** defines the array used by *mme_trksessionview_read()* to store track session view entries in memory.

trksessionview_entry_file_t

```
typedef struct {
    uint64_t  msid;
    uint32_t  reserved;
    char      *filename;
} trksessionview_entry_file_t;
```

The data structure **trksessionview_entry_file_t** carries information about tracks in a track session. It contains the following members:

Member	Type	Description
msid	**uint64_t**	The mediastore ID of the mediastore with the track.
reserved	**uint32-t**	Reserved for future use.
filename	**char**	The filename of the track.

TRKVIEW_READ_*

```
#define TRKVIEW_READ_FID  0x00000001
#define TRKVIEW_READ_FILE 0x00000002
```

The TRKVIEW_READ_* constants are used to set the type of information *mme_trksessionview_readx()* requests from the **`trksessionview`** table. Its value can be set to:

- TRKVIEW_READ_FID (0x00000001) — get the file IDs for the tracks
- TRKVIEW_READ_FILE (0x00000002) — get the file offsets for tracks

Events

None delivered.

Blocking and validation

This function doesn't block.

Returns:

≥**`0`**: the number of elements the function successfully read.

Success.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_trksessionview_get_info(), **`mme_trksessionview_info_t`**, *mme_trksessionview_metadata_get()*, *mme_trksessionview_writedb()*

mme_trksessionview_update()

Update the information in the **trksessionview** *table*

Synopsis:

```
#include <mme/mme.h>

int mme_trksessionview_update( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_trksessionview_update()* causes the MME to update the information for the current control context .

The **trksessionview** table stores static snapshots of track sessions at the time that they are set. The entries in this table do not, therefore, reflect changes to the database that have occurred since the track session was set. For example, files and metadata that were added to the database after the track session was set remain unknown to the track session.

To update the track session snapshot with the latest information provided by a concurrent synchronization, call *mme_trksessionview_update()*.

The track session view is accurate in memory as soon as *mme_trksessionview_update()* returns, regardless of whether the connection is synchronous our asynchronous. However, depending on how the MME is configured, the updates may or may not have been written to the **trksessionview** table. Only receipt of the MME_EVENT_TRKSESSIONVIEW_COMPLETE event confirms that the updates have been written to the **trksessionview** table.

If your system is configured *not* to automatically write track session view updates to the database (**<TrksessionViewAutoWrite>** set to **false**), you must call *mme_trksessionview_writedb()* to update the **trksessionview** table.

For both library-based and file-based track sessions, a call to *mme_trksessionview_update()* refreshes the pseudo-random order of the tracks in the track session.

For more information, see “Working with track sessions” in the chapter Playing Media of the *MME Developer’s Guide*.

Events

Client applications that call *mme_trksessionview_update()* can expect to see the following sequence of events:

1 MME_EVENT_TRKSESSIONVIEW_INVALID

2 MME_EVENT_TRKSESSIONVIEW_UPDATE — one or more times, if your system is configured to write updates to the database

3 MME_EVENT_TRKSESSIONVIEW_COMPLETE — when the track session snapshot is written to the **`trksessionview`** table

Blocking and validation

This function doesn't block.

Returns:

≥**`0`** Success: the MME updated the state of the current track session.

`-1` An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_play_resume_msid(), *mme_trksession_get_info()*, *mme_set_msid_resume_trksession()*, *mme_trksession_resume_state()*. *mme_trksession_save_state()*

mme_trksessionview_writedb()

Write the MME track session view to the database

Synopsis:

```
#include <mme/mme.h>

int mme_trksessionview_writedb( mme_hdl_t *hdl );
```

Arguments:

hdl An MME connection handle.

Library:

mme

Description:

The function *mme_trksessionview_writedb()* writes the current track session view to the **trksessionview** table in the MME database.

When the MME is configured to keep track session views in memory, it does not write the track session view to the MME database unless it is instructed to do so by a call to *mme_trksessionview_writedb()*. This function can be used to save track session views when the system is idle, or at system shutdown.

Events

This function delivers the event MME_EVENT_TRKSESSIONVIEW_COMPLETE when it has finished writing the track session view to the database. If the track session view has already been written to the database, this function will not write it a second time, but will nevertheless deliver MME_EVENT_TRKSESSIONVIEW_COMPLETE.

Blocking and validation

This function performs no validations, and returns immediately.

Returns:

>0 Success

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_trksessionview_get_info(), **mme_trksessionview_info_t**, *mme_trksessionview_metadata_get()*, *mme_trksessionview_readx()*

mme_video_get_angle_info()

Get the video angle

Synopsis:

```
#include <mme/mme.h>

int mme_video_get_angle_info ( mme_hdl_t *hdl,
                               uint64_t title,
                               mm_video_angle_info_t *info );
```

Arguments:

hdl An MME connection handle.

title The video title for which angle information is requested.

info A pointer to a **mm_video_angle_info_t** structure that carries information about the video angle.

Library:

mme

Description:

The function *mme_video_get_angle_info()* gets the video angle for an MME control context, and places it in the data structure **mm_video_angle_info_t**.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Examples:

```
mm_video_angle_info_t info;
unit_64 title = 1;

rc = mme_video_get_angle_info( mmehdl, title, &info );
if ( rc == 0 ) {
    sprintf( output, "Total: %d; Current: %d",
        info.total, info.current );
} else {
    sprintf( output, "Error getting video angle info: %s (%d).",
```

```
            strerror( errno ), errno );
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_audio_info(), *mme_video_get_info()*, *mme_video_get_status()*, *mme_video_get_subtitle_info()*, *mme_video_set_angle()*, *mme_video_set_audio()*, *mme_video_set_properties()*, *mme_video_set_subtitle()*

mme_video_get_audio_info()

Get the audio information for video playback

Synopsis:

```
#include <mme/mme.h>

int mme_video_get_audio_info ( mme_hdl_t *hdl,
                               uint64_t title,
                               mm_video_audio_info_t *info );
```

Arguments:

hdl An MME connection handle.

title The video title for which you want to get the audio information.

info A pointer to a **mm_video_audio_info_t** structure that carries information about the title's audio settings.

Library:

mme

Description:

The function *mme_video_get_audio_info()* gets information about audio settings for video playback in a control context and places it in the structure **mm_video_audio_info_t** described in this reference.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Examples:

From **mmecli**:

```
mm_video_angle_info_t info;
unit_64 title = 1;

rc = mme_video_get_angle_info( mmehdl, title, &info );
if ( rc == 0 ) {
```

```
    sprintf( output, "Total: %d; Current: %d",
        info.total, info.current );
} else {
    sprintf( output, "Error getting video angle info: %s (%d).",
        strerror( errno ), errno );
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), *mme_video_get_info()*, *mme_video_get_status()*, *mme_video_get_subtitle_info()*, *mme_video_set_angle()*, *mme_video_set_audio()*, *mme_video_set_properties()*, *mme_video_set_subtitle()*

mme_video_get_info()

Get information about a video.

Synopsis:

```
#include <mme/mme.h>

int mme_video_get_info( mme_hdl_t *hdl,
                        mm_video_info_t *info );
```

Arguments:

hdl An MME connection handle.

msid A pointer to a **mm_video_info_t** structure that *mme_video_info()* can fill with the information about the current video.

Library:

mme

Description:

The function *mme_video_get_info()* gets information about a video, including:

- aspect ratio
- dimensions (height and width)
- display mode
- capture format

For information about the structure **mm_video_info_t**, see **mm_video_info_t** in this reference.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success: the MME retrieved the information about the video.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_status(), mme_video_get_subtitle_info(), mme_video_set_angle(), mme_video_set_audio(), mme_video_set_properties(), mme_video_set_audio()

mme_video_get_status()

Get the video status

Synopsis:

```
#include <mme/mme.h>

int mme_video_get_status ( mme_hdl_t *hdl,
                           mm_video_status_t *status );
```

Arguments:

hdl An MME connection handle.

status A pointer to a **mm_video_status_t** structure that the function fills in with information about the video playback status.

Library:

mme

Description:

The function *mme_video_get_status()* gets status information for video playback of any format. The MME indicates that there has been a change in video status by sending a MME_EVENT_VIDEO_STATUS event.

To get DVD device status, use *mme_dvd_get_status()*.

For more information about video dimensions and aspect ratio, see **mm_video_info_t** in this reference.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Examples:

From **mmecli**:

```
mm_video_status_t status;
```

```
rc = mme_video_get_status( mmehdl, &status );
if ( rc == -1 ) {
    sprintf( output, "Error getting video status: %s (%d).",
        strerror( errno ), errno );
} else {
    sprintf( output, "Size: %ux%u; Aspect Ratio: %ux%u.",
        status.width, status.height,
        status.aspect_ratio.w,status.aspect_ratio.h );
}
```

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), *mme_video_get_audio_info()*, *mme_video_get_info()*, *mme_video_get_subtitle_info()*, *mme_video_set_angle()*, *mme_video_set_audio()*, *mme_video_set_properties()*, *mme_video_set_audio()*

mme_video_get_subtitle_info()

Get subtitle information for a video title

Synopsis:

```
#include <mme/mme.h>

int mme_video_get_subtitle_info ( mme_hdl_t *hdl,
                                  uint64_t title,
                                  mm_video_subtitle_info_t *info );
```

Arguments:

hdl An MME connection handle.

title The video title for which you want to get the subtitle information.

info A pointer to a **mm_video_subtitle_info_t** structure that carries information about the video subtitles.

Library:

mme

Description:

The function *mme_video_get_subtitle_info()* gets information about the subtitle for video playback for a control context and places it in the structure **mm_video_subtitle_info_t**.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥0 Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_info(), mme_video_get_status(), mme_video_set_angle(), mme_video_set_audio(), mme_video_set_properties(), mme_video_set_audio()

mme_video_set_angle()

Set the video angle

Synopsis:

```
#include <mme/mme.h>

int mme_video_set_angle ( mme_hdl_t *hdl,
                          uint64_t title,
                          int index );
```

Arguments:

hdl An MME connection handle.

title The video title on which to set the angle.

index An index to a desired angle from the array filled in by a previous call to *mme_video_get_angle_info()*. 0 (zero) points to the first available choice.

Library:

mme

Description:

The function *mme_video_set_angle()* sets the video angle for video playback. Before calling this function, use *mme_video_get_angle_info()* to get the current video angle.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No

continued...

Safety	
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_info(), mme_video_get_status(), mme_video_get_subtitle_info(), mme_video_set_audio(), mme_video_set_properties(), mme_video_set_audio()

mme_video_set_audio()

Set the audio stream for video playback

Synopsis:

```
#include <mme/mme.h>

int mme_video_set_audio ( mme_hdl_t *hdl,
                          uint64_t title,
                          int index );
```

Arguments:

hdl An MME connection handle.

title The title to set the audio for.

index An index to a desired audio stream from the array filled in by a previous call to *mme_video_get_audio_info()*. 0 (zero) points to the first available choice. A -1 in this parameter disables audio.

Library:

mme

Description:

The function *mme_video_set_audio()* sets the audio stream for video playback in a control context.

The MME 1.1.0 release does not support dynamic setting of audio attributes during video playback. These attributes should be set before starting playback. See also the data structure **mm_video_audio_info_t**.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

`mm_video_audio_info_t`, *mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_info(), mme_video_get_status(), mme_video_get_subtitle_info(), mme_video_set_angle(), mme_video_set_properties(), mme_video_set_audio()*

mme_video_set_properties()

Set the properties of a video.

Synopsis:

```
#include <mme/mme.h>

int mme_video_set_properties( mme_hdl_t *hdl,
                              mm_video_properties_t *props );
```

Arguments:

hdl An MME connection handle.

props The pointer to the structure with the properties to set for the video.

Library:

mme

Description:

The function *mme_video_set_properties()* sets video properties, and places the data in the structure **mm_video_properties_t** described in this reference. The properties set by *mme_video_set_properties()* include:

- dimensions (height and width)
- display mode
- zoom mode

Currently **io-media-generic** only supports setting the video source and destination (the *source* and *dest* members of the **mm_video_properties_t** structure). Other **io-media** variants may support other capabilities.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success: the MME set the video properties.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_info(), mme_video_get_status(), mme_video_get_subtitle_info(), mme_video_set_angle(), mme_video_set_audio(), mme_video_set_audio()

mme_video_set_subtitle()

Set the subtitle for video playback

Synopsis:

```
#include <mme/mme.h>

int mme_video_set_subtitle ( mme_hdl_t *hdl,
                             uint64_t title,
                             int index );
```

Arguments:

hdl An MME connection handle.

title The title to set the subtitle for.

index An index to a desired subtitle from the array filled in by a previous call to *mme_video_get_subtitle_info()*. 0 (zero) points to the first available choice. A **-1** in this parameter disables subtitles.

Library:

mme

Description:

The function *mme_video_set_subtitle()* sets the subtitles for video playback a control context. Before calling this function, use *mme_get_subtitle_info()* to get the available subtitles for the video.

Events

None delivered.

Blocking and validation

This function blocks on **io-media**. It returns only when it has completed.

Returns:

≥**0** Success.

-1 An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_video_get_angle_info(), mme_video_get_audio_info(), mme_video_get_info(), mme_video_get_status(), mme_video_get_subtitle_info(), mme_video_set_angle(), mme_video_set_audio(), mme_video_set_properties()

mme_zone_create()

Create an output zone

Synopsis:

```
#include <mme/mme.h>

int mme_zone_create( mme_hdl_t *hdl,
                     const char *name,
                     uint64_t *zoneid );
```

Arguments:

hdl	An MME connection handle.
name	A pointer to the zone name.
zoneid	The zone ID returned by the function.

Library:

mme

Description:

The function *mme_zone_create()* creates an output zone. It returns the ID of the new zone it created.

The MME uses zones to manage output. Zones can be attached to a control context or detached from a control context. The MME sends playback from a control context only to the zones attached to that control context. For example, in an automobile with two zones: “driver” and “passengers”, the zone “passengers” could be attached to a control context playing a video, while the zone “driver” would not be attached. A DVD-video played back in the control context would be available only in the zone “passengers”, but not in the zone “driver”.

Events

None delivered.

Blocking and validation

This function executes to completion.

Returns:

≥**0**	Success: the ID of the created output zone.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_output_set_permanent(), mme_play_attach_output(), mme_play_detach_output(), mme_play_get_zone(), mme_play_set_zone(), mme_zone_delete()

mme_zone_delete()

Delete an output zone

Synopsis:

```
#include <mme/mme.h>

int mme_zone_delete( mme_hdl_t *hdl,
                     uint64_t *zoneid );
```

Arguments:

hdl	An MME connection handle.
zoneid	The ID of the zone to be deleted.

Library:

mme

Description:

The function *mme_zone_delete()* deletes the specified output zone. For more information about zones, see *mme_zone_create()*.

Events

None delivered.

Blocking and validation

This function runs to completion.

Returns:

≥**0**	Success.
-1	An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

See also:

mme_output_set_permanent(), mme_play_attach_output(), mme_play_detach_output(), mme_play_get_zone(), mme_play_set_zone(), mme_zone_create()

Chapter 2

MME Events

In this chapter. . .

MME events are like other QNX Neutrino events. They are signals or pulses used to notify a client application thread that a particular condition has occurred. Unlike signals and pulses, events can be used to carry data.

This chapter includes:

- About MME events
- MME event classes
- MME event data
- MME general events

For information about other types of MME events, see the following chapters in this reference:

- MME Synchronization Events
- MME Playback Events
- MME Media Copy and Ripping Events
- MME Metadata Events

For more information about events in general, see the *QNX Neutrino Programmer's Guide*.

About MME events

MME events are associated with the control contexts through which the client application is connected to the MME. Each client connected to a control context has its own event queue to which the MME delivers events.

To receive events from the MME, your client application must:

- after each new connection to the MME, call the function *mme_register_for_events()* to register for events
- call the function *mme_get_event()* when it needs to retrieve the MME events.

See also the chapter Starting Up and Connecting to the MME in the *MME Developer's Guide*.

MME event classes

MME events are divided into classes, which **`mme_event_class_t`** defines as bitmasks. Its values are described in the table below.

Constant	Value	Description
MME_EVENT_CLASS_PLAY	0x0001	See the chapter MME Playback Events.
MME_EVENT_CLASS_SYNC	0x0002	See the chapter MME Synchronization Events.
MME_EVENT_CLASS_COPY	0x0004	See the chapter MME Media Copy and Ripping Events.
MME_EVENT_CLASS_GENERAL	0x0008	See “Events not specified in the other classes” below.
MME_EVENT_CLASS_METADATA	0x0010	See the chapter MME Metadata Events.
MME_EVENT_CLASS_ALL	0xFFFF	All events.

The MME event classes are bitmasks. They can be used together with an OR operator to register for several events at once. For example, to register for *playback* and *synchronization* events call the function *mme_register_for_events()* as follows:

```
mme_register_for_events( hdl,
                         MME_EVENT_CLASS_PLAY | MME_EVENT_CLASS_SYNC,
                         event );
```

The client application can register each of its connections for any or all of these classes, as required.

MME event data

Event data is delivered in the following structures:

- **mme_copy_error_t**
- **mme_event_t**
- **mme_event_default_language_t**
- **mme_event_metadata_image_t**
- **mme_event_metadata_info_t**
- **mme_event_metadata_licensing_t**
- **mme_event_queue_size_t**
- **mme_event_type_t**
- **mme_first_fid_data_t**
- **mme_folder_sync_data_t**

- **mme_ms_update_data_t**
- **mme_play_command_error_t**
- **mme_play_error_t**
- **mme_play_error_track_t**
- **mme_sync_data_t**
- **mme_sync_error_t**
- **mme_trackchange_t**
- **mm_warning_info_t**

For more information about the structures **mme_sync_data_t** and **mme_first_fid_data_t**, and the **mme_*_error_t** structures, see the relevant sections below.

mme_copy_error_t

```
typedef struct mme_copy_error {
   uint32_t        type;
   uint64_t        cqid;
   uint32_t        reserved;
   union {
       uint64_t    value;
       uint64_t    msid;
   };
} mme_copy_error_t;
```

The structure **mme_copy_error_t** carries media copying and ripping error data. Its members are described in the table below:

Member	Type	Description
type	**uint32_t**	Type of media copying and ripping error.
cqid	**uint64_t**	The copy queue ID.
reserved	**uint32_t**	Reserved for internal use.
union	**uint64_t**	Either *value*: copy queue ID, or *msid*: mediastore ID.

mme_event_t

```
typedef struct _mme_event {
    mme_event_type_t  type;
    size_t            size;
    char              data[0];
} mme_event_t;
```

The structure **mme_event_t** is described in the table below:

Member	Type	Description
type	**mme_event_type_t**	The event type.
size	**size_t**	The size, in bytes of the event data.
data	**char**	The event data.

mme_event_default_language_t

```
#include <mme/types.h>

typedef struct s_mme_default_language_event {
    int        error;
    const char language[1];
} mme_event_default_language_t;
```

The data structure **mme_event_default_language_t** carries information delivered with a MME_EVENT_DEFAULT_LANGUAGE event, including the result of the last attempt to set the default language, and a NULL terminated string indicating the current default language. It includes at least the members described in the table below:

Member	Type	Description
error	**int**	The result of the last request; this member is set to EOK on success.
language	**const char**	A NULL terminated string that indicates the current default language.

mme_event_metadata_image_t

```
typedef struct s_mme_metadata_image_event {
    uint64_t                mdinfo_irid;
    int32_t                 error;
    uint32_t                reserved;
    mme_metadata_image_url_t url;
} mme_event_metadata_image_t;
```

The data structure **mme_event_metadata_image_t** carries data for the MME_EVENT_METADATA_IMAGE event. It includes at least the members listed in the table below:

Member	Type	Description
mdinfo_irid	**uint64_t**	A metadata image request identifier.

continued...

Member	Type	Description
error	**int**	The *errno* returned with the request; set to EOK on success.
reserved	**uint32_t**	Reserved for internal use.
url	**mme_metadata_image_url_t**	The structure with the URL location for the image.

mme_event_metadata_info_t

```
typedef struct s_mme_metadata_info_event {
    uint64_t            mdinfo_rid;
    int32_t             error;
    uint32_t            reserved;
    mme_metadata_info_t metadata;
} mme_event_metadata_info_t;
```

The data structure **mme_event_metadata_info_t** carries data for the MME_EVENT_METADATA_INFO event. It includes at least the members listed in the table below:

Member	Type	Description
mdinfo_irid	**uint64_t**	A metadata information request identifier.
error	**int**	The *errno* returned with the request; set to EOK on success.
reserved	**uint32_t**	Reserved for internal use.
metadata	**mme_metadata_info_t**	The structure with the metadata.

mme_event_metadata_licensing_t

```
typedef struct mme_event_metadata_licensing {
    uint64_t    msid;
    uint64_t    fid;
    char        license[32];
} mme_event_metadata_licensing_t;
```

The data structure **mme_event_metadata_licensing_t** carries metadata licensing data. It includes at least the members described in the table below:

Member	Type	Description
msid	**uint64_t**	During mediastore synchronizations, the mediastore ID.

continued...

Member	Type	Description
fid	**uint64_t**	During individual file synchronizations, the file ID.
license	**char**	The license agreement, up to 32 characters long.

mme_event_queue_size_t

```
typedef struct s_mme_event_queue_size {
    size_t  first_event;
    size_t  all_events;
} mme_event_queue_size_t;
```

The structure **mme_event_queue_size_t** carries data for the event MME_EVENT_BUFFER_TOO_SMALL event. It includes at least the members listed in the table below:

Member	Type	Description
first_event	**size_t**	The size, in bytes, of the first event.
all_events	**size_t**	The size, in bytes, of all the events in the queue.

mme_event_type_t

```
typedef enum mme_event_type {
...
    MME_EVENT_*
    ...
} mme_event_type_t;
```

The enumerated type **mme_event_type_t** defines the types of events delivered by the MME. For details, see the events described in this chapter.

mme_first_fid_data_t

```
mme_first_fid_data {
    uint64_t    fid;
    uint64_t    msid;
    uint64_t    timestamp;
    uint32_t    operation_id;
    uint32_t reserved;
} mme_first_fid_data_t;
```

The structure **mme_first_fid_data_t** carries the file ID (*fid*) and mediastore ID (*msid*) for the first file and mediastore found during synchronization. It has at least these members:

Member	Type	Description
fid	**uint64_t**	The file ID.
msid	**uint64_t**	The mediastore ID.
timestamp	**uint64_t**	During a first synchronization pass, the MME's internal timestamp from the **library** table *last_sync* field; set to 0 (zero) at the second synchronization pass.
operation_id	**uint32_t**	The ID of the synchronization process that delivers the event carrying **mme_first_fid_data_t**.
reserved	**uint32_t**	Reserved for internal use.

mme_folder_sync_data_t

```
typedef struct s_mme_folder_sync_data {
    uint64_t    msid;
    uint64_t    folderid;
    uint32_t    pass;
    uint32_t    num_files;
    uint32_t    num_folders;
    uint32_t    num_playlists;
    uint64_t    timestamp;
    uint32_t    operation_id;
    uint32_t reserved;
} mme_folder_sync_data_t;
```

The data structure **mme_folder_sync_data_t** carries event data for folder synchronizations. It contains at least the members described in the table below:

Member	Type	Description
msid	**uint64_t**	The ID of the mediastore with the folder being synchronized.
folderid	**uint64_t**	The ID of the folder being synchronized.
pass	**uint32_t**	The synchronization pass this event is for; uses the MME_SYNC_OPTION_PASS_* flags.
num_files	**uint32_t**	See event specific documentation.
num_folders	**uint32_t**	See event specific documentation.
num_playlists	**uint32_t**	The number of playlists added to the **playlist** file.
timestamp	**uint64_t**	A timestamp of the last synchronization (*last_sync* value) for items associated with the event that delivers this structure.

continued...

Member	Type	Description
operation_id	**uint32_t**	The operation ID.
reserved	**uint32_t**	Reserved for internal use.

mme_ms_update_data_t

```
typedef struct s_mms_ms_update_data {
    uint64_t msid;
    uint64_t added_filecount;
    uint64_t added_foldercount;
    uint32_t operation_id;
    uint32_t flags;
    uint64_t timestamp;
} mme_ms_update_data_t;
```

The data structure **mme_ms_update_data_t** carries data about files information copied during synchronizations. It is described in the table below:

Member	Type	Description
msid	**uint64_t**	The ID of the synchronized mediastore.
added_filecount	**uint64_t**	The number of file IDs (*fid*s) added to the MME database by this synchronization.
added_foldercount	**uint64_t**	The number of folders added to the MME database by this synchronization.
operation_id	**uint32_t**	The operation ID.
flags	**uint32_t**	The type of operation. See *flags* below.
timestamp	**uint64_t**	The time stamp to write in the *last_sync* column for items associated with the event that carries this data structure.

flags

The **mme_ms_update_data_t** member *flags* can have the following values:

flags	Meaning
0	Not a synchronization operation
MME_SYNC_OPTION_PASS_FILES	Synchronization pass 1
MME_SYNC_OPTION_PASS_METADATA	Synchronization pass 2

continued...

flags	Meaning
MME_SYNC_OPTION_PASS_PLAYLISTS	Synchronization pass 3

mme_play_command_error_t

```
typedef struct mme_play_command_error {
    uint32_t    command;
    uint32_t    button;
    uint64_t    fid;
} mme_play_command_error_t;
```

The structure **mme_play_command_error_t** carries the playback error types. Its members are described in the table below:

Member	Type	Description
command	**mme_command_type_t**	Command error data.
button	**uint32_t**	Button error data.
fid	**uint32_t**	The file ID of the file being accessed when the error occurs.

mme_play_error_t

```
typedef struct mme_play_error {
   uint32_t                  type;
   uint32_t                  reserved;
   union {
      uint64_t               value;
      uint64_t               fid;
      uint64_t               trksessionid;
      uint64_t               msid;
      uint64_t               outputid;
      mme_play_command_error_t  command_error;
      mme_play_error_track_t    track;
  };
} mme_play_error_t;
```

The structure **mme_play_error_t** carries playback error data. Its members are described in the table below:

Member	Type	Description
type	**uint32_t**	The type of playback error; see "Playback error events" in the chapter MME Playback Events.
reserved	**uint32_t**	Reserved for internal use.
value	**uint64_t**	The error value.
fid	**uint64_t**	The file ID.
trksessionid	**uint64_t**	The track session ID.
msid	**uint64_t**	The mediastore ID.
outputid	**uint64_t**	The output ID.
command_error_t	**mme_play_command_error_t**	The command error type.
mme_play_error_track_t	**mme_play_error_track_t**	The file ID or the offset of the track that generated the error.

mme_play_error_track_t

```
typedef struct mme_play_error_track {
    uint64_t                fid;
    uint64_t                offset;
} mme_play_error_track_t;
```

The data structure **mme_play_error_track_t** carries the file ID or the offset in the track session of the track where a playback error occurred. It contains the following members:

Member	Type	Description
fid	**uint64_t**	The file ID of the track where an error occurred; used for library-based track sessions.
offset	**uint64_t**	The offset of the track where an error occurred; used for file-based track sessions.

mme_sync_data_t

```
typedef struct mme_sync_data {
    uint64_t    msid;
    uint32_t    operation_id;
   uint32_t    reserved;
} mme_sync_data_t;
```

The structure **mme_sync_data_t** carries data for many synchronization events (MME_EVENT_MS_SYNC_*). It has these members:

Member	Type	Description
msid	**uint64_t**	The mediastore ID.
operation_id	**uint32_t**	The synchronization operation ID.
reserved	**uint32_t**	Reserved for internal use.

mme_sync_error_t

```
typedef struct mme_sync_error {
    uint32_t    type;
    uint32_t    operation_id;
    uint32_t    param;
    uint32_t    reserved;
    uint64_t    msid;
} mme_sync_error_t;
```

The structure **mme_sync_error_t** carries synchronization error data. Its members are described in the table below:

Member	Type	Description
type	**uint32_t**	The type of synchronization error.
operation_id	**uint32_t**	The synchronization operation ID.
param	**uint32_t**	Parameters for the synchronization.
reserved	**uint32_t**	Reserved for internal use.
msid	**uint64_t**	The mediastore ID.

mme_trackchange_t

```
typedef struct mme_trackchange {
   uint64_t    fid;
   uint64_t    fid_requested;
   uint64_t    offset;

} mme_trackchange_t;
```

The data structure **mme_trackchange_t** carries data for the MME_EVENT_TRACKCHANGE event. It is described in the table below:

Member	Type	Description
fid	**uint64_t**	The file ID of the track being played.
fid_requested	**uint64_t**	The file ID that was requested for playback. In most cases *fid* and *fid_requested* will have the same values. However, when playback occurs during a ripping operation, *fid* and *fid_requested* may be different, because the client application may request playback of a track from the source, such as a CDDA, but the MME will play the ripped destination file on the HDD.
offset	**uint64_t**	The current offset in the track session.

mm_warning_info_t

```
typedef struct s_mm_warning_info {
_Uint32t mm_warning; /* mm_warnings_t */
_Uint32t flags; /* mm_warning_flags_t */
} mm_warning_info_t;
```

The structure **mm_warning_info_t** carries information about the conditions that have caused a warning. It contains at least the members described in the table below.

Member	Type	Description
mm_warning	**mme_warnings_t**	The type of warning condition reported by**io-media**.
flags	**mme_warning_flags_t**	Information about the conditions that caused the warningPlayback warnings for **io-media**.

mm_warnings_t

The enumerated type **mm_warnings_t** defines the type of warning condition detected. Its values and the behaviors they define are described below:

- MM_WARNING_READ_TIMEOUT — playback of a partially ripped file is in danger of over-running the end of that file.

mm_warning_flags_t

The enumerated type **mm_warning_flags_t** defines the state of an MME operation associated with a warning. Its values and the behaviors they define are described below:

- MM_WARNING_FLAG_NONE — no user-perceivable warning condition exists.
- MM_WARNING_FLAG_AUDIBLE — playback has over-run the end of a partially ripped or copied file, and the user has encountered an audible gap.

MME general events

The MME delivers general events (MME_EVENT_CLASS_GENERAL) to the client application to indicate changes in status, or the result of an activity.

The MME general events are:

- MME_EVENT_AUTOPAUSECHANGED
- MME_EVENT_BUFFER_TOO_SMALL
- MME_EVENT_DEFAULT_LANGUAGEL
- MME_EVENT_NONE
- MME_EVENT_SHUTDOWN
- MME_EVENT_SHUTDOWN_COMPLETED
- MME_EVENT_USERMSG

MME_EVENT_AUTOPAUSECHANGED

The MME delivers the event MME_EVENT_AUTOPAUSECHANGED after it has changed the autopause mode for a specified control context.

To change the autopause mode for a control context, call the function *mme_setautopause()*.

Event data

The new autopause setting, in a **`uint64_t`**:

1 Enabled.

0 Disabled.

Database tables updated

No database tables are updated.

MME_EVENT_BUFFER_TOO_SMALL

The MME delivers the event MME_EVENT_BUFFER_TOO_SMALL to a client application when the client application's event buffer is too small to retrieve any events from the MME.

Event data

The size, in bytes, of the first event in the event buffer, in **`mme_event_queue_size_t.first_event`**, and the size of all the events in the event queue, in **`mme_event_queue_size_t.all_events`**.

Database tables updated

No database tables are updated.

MME_EVENT_DEFAULT_LANGUAGE

The MME function *mme_media_set_def_lang()* delivers the event MME_EVENT_DEFAULT_LANGUAGE to indicate that the default preferred language for a media item has been set.

Event data

The success or failure of the default preferred language update, and the preferred language, in **`mme_event_default_language_t`**.

The string in **`mme_event_default_language_t.language`** *always* indicates the current default preferred language. That is, if *mme_media_set_def_lang()* is unable to change the default language to the requested language, this string will indicate the preferred language before the function call was made (because it is still the *set* preferred language).

Database tables updated

No database tables are updated.

MME_EVENT_NONE

The MME delivers the event MME_EVENT_NONE to a client application when there are no events in the queue for the control context from which the client application requested events.

To request events, use the function *mme_get_event()*.

Event data

No data.

Database tables updated

No database tables are updated.

MME_EVENT_SHUTDOWN

The MME delivers the event MME_EVENT_SHUTDOWN to all control contexts after it receives a request to shut down. If your client application receives this event, it should inform the user that it is shutting down.

To shut down the MME, call the function *mme_shutdown()*.

The function *mme_shutdown()* returns immediately and shuts down MME threads in the background. This behavior means that the MME may deliver other events *after* it has delivered MME_EVENT_SHUTDOWN. When all MME threads have shut down, the MME delivers the event MME_EVENT_SHUTDOWN_COMPLETED.

Event data

No data.

Database tables updated

No database tables are updated.

MME_EVENT_SHUTDOWN_COMPLETED

The MME delivers the event MME_EVENT_SHUTDOWN_COMPLETED to all control contexts to indicate that it has completed its shutdown preparations. Playback, ripping and synchronization operations have been stopped.

Event data

No data.

Database tables updated

No database tables updated.

CAUTION: Operations attempted with the MME after it has delivered the MME_EVENT_SHUTDOWN_COMPLETED event may produce unexpected results and compromise the integrity of your system. To use the MME after receiving the MME_EVENT_SHUTDOWN_COMPLETED event, you should terminate the MME, then start it again.

MME_EVENT_USERMSG

Not currently implemented.

Chapter 3

MME Synchronization Events

In this chapter. . .

MME events are like other QNX Neutrino events. They are signals or pulses used to notify a client application thread that a particular condition has occurred. Unlike signals and pulses, events can be used to carry data.

This chapter includes:

- Synchronization events
- Synchronization error events

For other information about other types of MME events, see the following chapters in this reference:

- MME Events
- MME Playback Events
- MME Media Copy and Ripping Events
- MME Metadata Events

For more information about events in general, see the *QNX Neutrino Programmer's Guide*.

Synchronization events

The MME delivers synchronization events (MME_EVENT_CLASS_SYNC) to the client application to indicate the status or result of a synchronization.

The MME synchronization events are:

- MME_EVENT_MS_DETECTION_DISABLED
- MME_EVENT_MS_DETECTION_ENABLED
- MME_EVENT_METADATA_LICENSING
- MME_EVENT_MS_1PASSCOMPLETE
- MME_EVENT_MS_2PASSCOMPLETE
- MME_EVENT_MS_3PASSCOMPLETE
- MME_EVENT_MS_STATECHANGE
- MME_EVENT_MS_SYNCCOMPLETE
- MME_EVENT_MS_SYNC_FIRST_EXISTING_FID
- MME_EVENT_MS_SYNCFIRSTFID
- MME_EVENT_MS_SYNC_FOLDER_COMPLETE
- MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE

- MME_EVENT_MS_SYNC_FOLDER_STARTED
- MME_EVENT_MS_SYNC_PENDING
- MME_EVENT_MS_SYNC_STARTED
- MME_EVENT_MS_UPDATE
- MME_EVENT_SYNCABORTED
- MME_EVENT_SYNC_ERROR
- MME_EVENT_SYNC_SKIPPED

MME_EVENT_MS_DETECTION_DISABLED

The MME synchronization event MME_EVENT_MS_DETECTION_DISABLED is for future use.

Event data

No event data is delivered.

Database table updated

No database tables are updated.

MME_EVENT_MS_DETECTION_ENABLED

The MME delivers the synchronization event MME_EVENT_MS_DETECTION_ENABLED when it has successfully read a path monitoring configuration, connected to a path monitoring system, and enabled device detection.

Event data

No event data is delivered.

Database table updated

The following table is updated:

- **`mediastores`** — entries in the database that were set to **`active`** or **`available`** are set to **`unavailable`**

After the MME delivers the event MME_EVENT_MS_DETECTION_ENABLED it may begin processing mediastore state changes and updating the **`mediastores`** table accordingly.

MME_EVENT_METADATA_LICENSING

The MME delivers the synchronization event MME_EVENT_METADATA_LICENSING when it uses a metadata service that has special licensing requirements, such as a requirement to display the service's logo. The MME delivers this event each time it begins using the metadata service.

Event data

This event delivers the following data, in **`mme_event_metadata_licensing_t`**, as follows:

- If the MME is synchronizing an entire mediastore, this event delivers the mediastore ID in **`mme_event_metadata_licensing_t.msid`**, with **`mme_event_metadata_licensing_t.fid`** set to 0 (zero).
- If the MME is performing synchronizations for individual files, this event delivers the file ID in **`mme_event_metadata_licensing_t.fid`**, with **`mme_event_metadata_licensing_t.msid`** set to 0 (zero).
- The licensing requirement, in **`mme_event_metadata_licensing_t.license`**.

Database table updated

No database tables are updated.

MME_EVENT_MS_1PASSCOMPLETE

The MME delivers the event MME_EVENT_MS_1PASSCOMPLETE when it has completed the first pass of file and folder synchronization between a mediastore and the MME library.

With the first synchronization pass the MME:

- sets the *valid* field to 0 (not valid) for all file IDs (*fid*s) that are in the library for the mediastore being synchronized.
- recursively scans the mediastore for files and folders

When it finds a file on the mediastore, the MME:

- checks if the file is in the library. If the file is in the library, the MME:
  - sets that file's *valid* field to 1 to indicate that the file entry is valid
  - compares the mediastore and library file dates and sizes to determine if the file needs to be resynchronized
  - if the file needs to be resynchronized, the MME sets the files *accurate* field to 0, to indicate to subsequent syncrhonization passes that the file's information needs to be updated
  - if the file does not need to be synchronized, the MME leaves the *accurate* field set to 1 so that the file will not be resynchronized by subsequent synchronization passes

- if the file isn't in the library, the MME:
  - adds the file to the library
  - sets the file's *valid* field to 1 and its *accurate* field to 0

When it finds a folder (directory) on the mediastore, the MME:

- adds the folder to the **`folders`** table
- synchronizes the folder by looking for files and folders inside it.

The MME adds files only if their extensions match the media support extensions defined by the in element **`<SyncFileMask>`** in the file configuration file **`mme.conf`**. For more information, see "Configurable file skipping: **`<SyncFileMask>`**" in the chapter Configuring Media Synchronizations in the *MME Configuration Guide*.

When the first synchronization pass is complete:

- the following MME table is updated and is accurate:
  - **`mediastores`**
- the **`library`** table and associated tables are updated, but are not guaranteed to be accurate. The following key information from the **`library`** is reliable:
  - mediastore ID
  - folder IDs
  - file names
  - file sizes
  - title (iPod synchronizations only)

When it has completed the first synchronization pass, the MME sets to 1:

- the *syncflags* field for the synchronized mediastore in the **`mediastores`** table
- the *synced* field for all synchronized folders in the **`folders`** table

Event data

The synchronization data, in **`mme_sync_data_t`**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

The following tables are updated:

- **`folders`**
- **`library`**

- **mediastores**
- **playlists**

MME_EVENT_MS_2PASSCOMPLETE

The MME delivers the event MME_EVENT_MS_2PASSCOMPLETE to inform the client application that it has completed the second pass of file and folder synchronization.

When the second synchronization pass is complete:

- the MME library metadata for the media store being synchronized is complete and accurate

When it has completed the second synchronization pass, the MME adds the value **2** to the value of:

- the *syncflags* field for the synchronized mediastore in the **mediastores** table
- the *synced* field for all synchronized folders in the **folders** table

Thus, when the MME has completed the second syncrhonization pass, the updated fields *syncflags* and *synced* in the **mediastores** and **folders** tables have the value 3.

Event data

The synchronization data, in **mme_sync_data_t**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

The following tables are updated:

- **folders** (*synced* column)
- **library**
- **library_*** (**library_artists**, **library_genres**, etc.)

MME_EVENT_MS_3PASSCOMPLETE

The MME delivers the event MME_EVENT_MS_3PASSCOMPLETE when it has completed the third pass of file and folder synchronization for a mediastore.

During the third synchronization pass the MME:

- compiles the playlist for the mediastore being synchronized
- updates the table **playlist** and, if required, **playlistdata** with the playlist information for the mediastore

When the third synchronization pass is complete the MME has accurate and complete playlists for the mediastore.

When it has completed the second synchronization pass, the MME adds the value 4 to the value of:

- the *syncflags* field for the synchronized mediastore, in the **`mediastores`** table
- the *synced* field for all synchronized folders, in the **`folders`** table

Thus, when the MME has completed the third syncrhonization pass, the updated fields *syncflags* and *synced* in the **`mediastores`** and **`folders`** tables have the value 7.

Event data

The synchronization data, in **`mme_sync_data_t`**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

The following tables are updated:

- **`folders`** (*synced* column)
- **`playlistdata`**
- **`playlists`**

MME_EVENT_MS_STATECHANGE

The MME delivers the event MME_EVENT_MS_STATECHANGE when it has detected that a mediastore state has changed. Mediastore state changes occur when a mediastore:

- is inserted, and changes state from “non-existent” or “unavailable” to “active” or “available”
- is removed, and changes state from “active” or “available” to “unavailable”
- is pruned and changes state to “non-existent”
- changes state from “active” to “available”, or vice versa

A mediastore state change indicates only the state of the mediastore in the system. It does not provide information about the state of the mediastore synchronization.

Event data

The mediastore state data, in **`mme_ms_statechange_t`**:

- the mediastore ID (*msid*) of the mediastore, in **`mme_ms_statechange_t.msid`**
- the old (previous) state of of the mediastore, in **`mme_ms_statechange_t.old_state`**
- the new (current) state of of the mediastore, in **`mme_ms_statechange_t.new_state`**
- the device type, in **`mme_ms_statechange_t.device_type`**
- the mediastore type, in **`mme_ms_statechange_t.storage_type`**

Database tables updated

The following table is updated:

- **`mediastores`**

MME_EVENT_MS_SYNCCOMPLETE

The MME delivers the event MME_EVENT_MS_SYNCCOMPLETE when it has successfully completed all requested synchronization passes for a mediastore.

When the synchronization is complete:

- the MME has finished all synchronization activities required for the mediastore
- all MME metadata and playlist information for the mediastore is current and accurate, and the client application can use the full library view

Different mediastore types may require different synchronization activities, including a different number of synchronization passes. If it checks for the MME_EVENT_MS_SYNCCOMPLETE event, the client application does not need to know the number of synchronization passes required for a media type. When it receives the MME_EVENT_MS_SYNCCOMPLETE event the client application knows that the MME has successfully completed all requested synchronization activity required for the mediastore.

Receiving the event MME_EVENT_MS_SYNCCOMPLETE does not mean that there will necessarily be files to play. For example, requesting only the second synchronization pass won't populate the MME tables with the minimum information needed to build track sessions and playing tracks.

When the MME synchronizes prunable mediastores that it has synchronized earlier, the MME may clean up unused metadata in its database. This clean up may take up to several seconds, depending on the size of the MME database, and cause a corresponding delay between delivery of the MME_EVENT_MS_*PASSCOMPLETE event and delivery of the MME_EVENT_MS_SYNCCOMPLETE event. For more information, see "Database clean up during synchronization" in the chapter Synchronizing Media of the *MME Developer's Guide*.

Event data

The synchronization data, in **`mme_sync_data_t`**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

The following tables are updated:

- **`folders`**
- **`library`**
- **`library_*`** (**`library_artists`**, **`library_genres`**, etc.).
- **`mediastores`**.
- **`playlistdata`**
- **`playlists`**

For more details, see the information for the specific synchronization passes.

MME_EVENT_MS_SYNC_FIRST_EXISTING_FID

The MME delivers the event MME_EVENT_MS_SYNC_FIRST_EXISTING_FID to inform the client application that it has a track or file that it can begin playing. It delivers this event under the following conditions:

- this is the first pass of a mediastore synchronization
- the MME has found the first playable track or file

Unlike MME_EVENT_MS_SYNCFIRSTFID, whose delivery confirms that all items in the database are valid, MME_EVENT_MS_SYNC_FIRST_EXISTING_FID only informs the client application that a playable file has been found. If the synchronization operation is pruning files from the database, there is no guarantee that all items in the database are valid.

The MME delivers this event on all (initial and subsequent) first synchronization passes of a mediastore, and when a new file has been synchronized during file synchronization (with *mme_sync_file()*).

Event data

The file ID (*fid*) and the mediastore ID (*msid*) of the first playable track or file:

- the ID of the mediastore being synchronized, in **`mme_first_fid_data_t.msid`**
- the file ID of the first playable track on this mediastore, in **`mme_first_fid_data_t.fid`**
- during a first synchronization pass, the MME's internal timestamp from the **`library`** table *last_sync* field, in **`mme_first_fid_data_t.timestamp`**; set to 0 (zero) at the second synchronization pass
- the ID of the synchronization process that delivers the event, in **`mme_first_fid_data_t.operation_id`**

Database tables updated

No database tables are updated.

MME_EVENT_MS_SYNCFIRSTFID

The MME delivers the event MME_EVENT_MS_SYNCFIRSTFID to:

- inform the client application that it has a track or file that it can begin playing
- confirm to the client application that all items in the database are valid

The MME delivers this event under the following conditions:

- this is the first pass of a mediastore synchronization
- the MME has found the first playable track or file
- if files must be removed from the database, *after* the MME has completed removal of these files

The MME delivers this event on all (initial and subsequent) first synchronization passes of a mediastore, and when a new file has been synchronized during file synchronization (with *mme_sync_file()*).

Event data

The file ID (*fid*) and the mediastore ID (*msid*) of the first playable track or file, in **`mme_first_fid_data_t`**:

- the ID of the mediastore being synchronized, in **`mme_first_fid_data_t.msid`**
- the file ID of the first playable track on this mediastore, in **`mme_first_fid_data_t.fid`**

- the MME's internal timestamp from the **`library`** table *last_sync* field, in **`mme_first_fid_data_t.timestamp`**
- the ID of the synchronization process that delivers the event, in **`mme_first_fid_data_t.operation_id`**

Database tables updated

No database tables are updated.

MME_EVENT_MS_SYNC_FOLDER_COMPLETE

The MME delivers the event MME_EVENT_MS_SYNC_FOLDER_COMPLETE when it completes a *non-recursive* synchronization of all files in a folder, and when the child folders in that folder have been enumerated.

Event data

The event data delivered differs between the first and second synchronization passes:

- first synchronization pass — **`mme_folder_sync_data_t`**:
  - if the folder is new or has changed since the last synchronization, **`mme_folder_sync_data_t.num_files`** with the number of files, and **`mme_folder_sync_data_t.num_folders`** with the number of child folders in the synchronized folder
  - if the folder is unchanged since the last synchronization, **`mme_folder_sync_data_t.num_files`** and **`mme_folder_sync_data_t.num_folders`** se to 0 (zero)
- second synchronization pass — the number of files that have changed since the last synchronization, in **`mme_folder_sync_data_t.num_files`**; and **`mme_folder_sync_data_t.num_folders`** set to 0.

Database table updated

The following table is updated:

- **`folders`** — the synchronization flags are set

MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE

The MME delivers the event MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE when it completes a *recursive* synchronization of all files and child folders in a folder. It does *not* deliver this event when is completes a non-recursive synchronization of a folder.

Event data

`mme_folder_sync_data_t.num_folders` with the number of synchronized child folders in this folder, and **`mme_folder_sync_data_t.num_files`** set to 0 (zero).

Database table updated

The following table is updated:

- **`folders`** — the synchronization flags are set

MME_EVENT_MS_SYNC_FOLDER_STARTED

The MME delivers the event MME_EVENT_MS_SYNC_FOLDER_STARTED when it begins synchronization of a folder; specifically, delivery is:

- first synchronization pass — after the MME has inserted the folder information in its database
- second synchronization pass — before the second pass begins on the contents of the folder

Event data

`mme_folder_sync_data_t`, with the number of files and the number of child folders in the folder being synchronized set to 0 (zero).

Database table updated

The following table is updated:

- **`folders`**

MME_EVENT_MS_SYNC_PENDING

The MME delivers the event MME_EVENT_MS_SYNC_PENDING when it has placed a mediastore on the synchronization pending list because it does not have a synchronization thread available to perform the synchronization.

Event data

The synchronization data, in **`mme_sync_data_t`**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation.

Database tables updated

No database tables are updated.

MME_EVENT_MS_SYNC_STARTED

The MME delivers the event MME_EVENT_MS_SYNC_STARTED when it has started synchronization of a mediastore.

Event data

The synchronization data, in **mme_sync_data_t**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

The following tables are updated:

- **mediastores**

MME_EVENT_MS_UPDATE

The MME delivers the event MME_EVENT_MS_UPDATE when it writes new data to the MME database during:

- a mediastore synchronization
- other operations that update the **library** table, or a **library_*** table

Delivery of MME_EVENT_MS_UPDATE other than during a synchronization operation indicates one of the following:

- an external CD changer that manages its own track sessions is playing a track that isn't in the **library**
- a ripping operation has added metadata for a file to the **library** table, or a **library_*** table (**<UpdateMetadata enable="true">**)
- the MME uses metadata from the **nowplaying** table to update metadata in a **library** table, or a **library_*** table (**<UpdateLibraryFromNowplaying enabled="true"/>**)

Event data

This event carries data about the operation, in **mme_ms_update_data_t**.

Database tables updated

The MME updates different tables, depending on the operation that delivers the event. The type of operation is indicated by the value of **mme_ms_update_data_t.flags** carried by the event.

flag=0 **library**
library_*

flag=MME_SYNC_OPTION_PASS_FILES

folders

library
mediastores
playlists

flag=MME_SYNC_OPTION_PASS_METADATA

folders
library
library_*
mediastores

flag=MME_SYNC_OPTION_PASS_PLAYLISTS

mediastores
playlists
playlistsdata

MME_EVENT_SYNCABORTED

The MME delivers the event MME_EVENT_SYNCABORTED when a synchronization operation is aborted. When this event is delivered, the mediastore is partially synchronized with the library. The extent of this synchronization can vary greatly, depending on how far the synchronization had progressed.

Event data

The synchronization data, in **mme_sync_data_t**:

- the ID of the mediastore being synchronized
- the operation ID (0 for non-directed synchronizations) for the synchronization operation

Database tables updated

No database tables are updated.

MME_EVENT_SYNC_ERROR

The MME delivers the event MME_EVENT_SYNC_ERROR when an error occurs during a synchronization operation. The cause of the error is carried in the event data.

Event data

The synchronization error code, in **mme_sync_error_t**.

Database tables updated

No database tables are updated.

MME_EVENT_SYNC_SKIPPED

The MME delivers the event MME_EVENT_SYNC_SKIPPED when it has detected the insertion of a mediastore, but has not automatically started synchronization of this mediastore. When the client application receives this event it can request a synchronization of the mediastore.

This event is delivered if a mediastore is inserted into the system and any of the following conditions is true:

- The MME is configured to *not* automatically synchronize mediastores.
- The MME is configured to automatically synchronize mediastores, but the mediastore is of a type (iPod) that the MME does not automatically synchronize unless expressly configured to do so; that is, the **<ipod>**/**<auto_sync>** *permitted* attribute is set to **false**.
- An **ievent** plugin has indicated that synchronization should not proceed. The conditions under which this situation could occur are:
  - the system is configured to accept no more than a specified number of mediastores (the **<MaxMediastores>** configuration element has been set)
  - a new mediastore is inserted and the MME is unable to remove enough mediastore data to permit the synchronization of the current media store

Event data

The ID of the inserted mediastore, in a **uint64_t**.

Database tables updated

No database tables are updated.

Synchronization error events

The MME synchronization error events are defined by the enumerated type **mme_sync_error_type_t**:

```
typedef enum mme_sync_error_type {
    ...
    MME_SYNC_ERROR_*
    ...
} mme_sync_error_type_t;
```

The MME synchronization error events are:

- MME_SYNC_ERROR_MEDIABUSY
- MME_SYNC_ERROR_NETWORK
- MME_SYNC_ERROR_FOLDER_LIMIT
- MME_SYNC_ERROR_LIB_LIMIT

- MME_SYNC_ERROR_NOTSPECIFIED.
- MME_SYNC_ERROR_READ
- MME_SYNC_ERROR_UNSUPPORTED
- MME_SYNC_ERROR_USERCANCEL

MME_SYNC_ERROR_MEDIABUSY

The MME delivers the event MME_SYNC_ERROR_MEDIABUSY after it fails to start start synchronization of a mediastore because the mediastore was being used by a process, such as playback, that has higher priority than synchronization.

Event data

The ID of the skipped mediastore, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_NETWORK

The MME delivers the event MME_SYNC_ERROR_NETWORK when it is unable to complete a synchronization because of a network error.

Event data

The ID of the mediastore that was not synchronized, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_FOLDER_LIMIT

The MME delivers the event MME_SYNC_ERROR_FOLDER_LIMIT during the first synchronization pass, when it has reached the configured maximum number of items (files and folders) permitted in a folder. This error event does not indicate a terminal event. It informs the client application that:

- the number of items it has synchronized in the current folder has reached the maximum configured by the **`<MaxFolderItems>`** element in the MME configuration file
- the synchronization operation will synchronize no more items from this folder
- synchronization will proceed normally for the rest of the mediastore, updating metadata and the *last_sync* column

Event data

This event carries:

- the operation ID of the synchronization operation, in **mme_ms_update_data_t.operation_id**
- the ID of the mediastore being synchronized when the limit is reached, in **mme_ms_update_data_t.msid**.
- the folder ID of the folder where the configured limit was reached, in **mme_sync_error.param**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_LIB_LIMIT

The MME delivers the event MME_SYNC_ERROR_LIB_LIMIT during the first synchronization pass, when it can add no more entries to the **library** table. This error event does not indicate a terminal event. It informs the client application that:

- the number of entries in the **library** table has reached the limit set by *max_lib_entries*
- the synchronization operation will not add more entries to the **library** table
- synchronization will proceed normally for the mediastore, updating metadata and the *last_sync* column for entries in the **library** table

Event data

This event carries:

- the operation ID of the synchronization operation, in **mme_ms_update_data_t.operation_id**
- the ID of the mediastore being synchronized when the limit is reached, in **mme_ms_update_data_t.msid**.
- the entry limit reached by the mediastore, in **mme_sync_error.param**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_NOTSPECIFIED

The MME delivers the event MME_SYNC_ERROR_NOTSPECIFIED at any time during a synchronization process that it must stop synchronization due to an error not covered by the other error events.

Event data

The ID of the mediastore that was not synchronized, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_READ

The MME delivers the event MME_SYNC_ERROR_READ when it encounters a read error that prevents the mediastore from being synchronized. Read errors can be caused by scratched disks, or other similar faults in the mediastore.

Event data

The ID of the mediastore with the problem, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_UNSUPPORTED

The MME delivers the event MME_SYNC_ERROR_UNSUPPORTED when it is unable to start a synchronization because it does not support the mediastore format.

Event data

The ID of the mediastore that was not synchronized, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_SYNC_ERROR_USERCANCEL

The MME delivers the event MME_SYNC_ERROR_USERCANCEL when it stopped synchronization of mediastore in response to a cancellation request from the client application.

Event data

The ID of the mediastore that was not synchronized, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

Chapter 4

MME Playback Events

In this chapter. . .

MME events are like other QNX Neutrino events. They are signals or pulses used to notify a client application thread that a particular condition has occurred. Unlike signals and pulses, events can be used to carry data.

This chapter includes:

- Playback events
- Playback error events

For other information about other types of MME events, see the following chapters in this reference:

- MME Events
- MME Synchronization Events
- MME Media Copy and Ripping Events
- MME Metadata Events

For more information about events in general, see the *QNX Neutrino Programmer's Guide*.

Playback events

The MME delivers playback events (MME_EVENT_CLASS_PLAY) to the client application to indicate the status or result of a playback activity.

The MME playback events are:

- MME_EVENT_DVD_STATUS
- MME_EVENT_FINISHED
- MME_EVENT_FINISHED_WITH_ERROR
- MME_EVENT_MEDIA_STATUS
- MME_EVENT_NEWOUTPUT
- MME_EVENT_NOWPLAYING_METADATA
- MME_EVENT_OUTPUTATTRCHANGE
- MME_EVENT_OUTPUTREMOVED
- MME_EVENT_PLAYAUTOPAUSED
- MME_EVENT_PLAY_ERROR
- MME_EVENT_PLAYLIST
- MME_EVENT_PLAYSTATE

- MME_EVENT_PLAY_WARNING
- MME_EVENT_RANDOMCHANGE
- MME_EVENT_REPEATCHANGE
- MME_EVENT_SCANMODECHANGE
- MME_EVENT_TIME
- MME_EVENT_TRACKCHANGE
- MME_EVENT_TRKSESSION
- MME_EVENT_TRKSESSIONVIEW_COMPLETE
- MME_EVENT_TRKSESSIONVIEW_INVALID
- MME_EVENT_TRKSESSIONVIEW_UPDATE
- MME_EVENT_VIDEO_STATUS

MME_EVENT_DVD_STATUS

The MME delivers the event MME_EVENT_DVD_STATUS when there are changes to the status of a DVD playback. These changes can be to a DVD:

- title
- chapter
- domain
- forbiddden UOP

Event data

The DVD status, in **`mm_dvd_status_event_t`**.

Database tables updated

No database tables are updated.

MME_EVENT_FINISHED

The MME delivers the event MME_EVENT_FINISHED when it has finished playing a track session and repeat mode is turned off so that no other playback will occur automatically.

Event data

The ID of the track session, in a **`uint64_t`**.

Database tables updated

The following tables are updated:

- **`trksessions`**

MME_EVENT_FINISHED_WITH_ERROR

The MME delivers the event MME_EVENT_FINISHED_WITH_ERROR when it has consecutively failed to initiate playback of the number of tracks defined by the **`<ConsecutivePlayErrorsBeforeStop>`** configuration element. It delivers this event to indicate that the current track session is not playable, and that it requires input from the client application to continue.

When the client application receives an MME_EVENT_FINISHED_WITH_ERROR event, it should assume that the current track session can't be played to completion, and take appropriate action. It may choose to call *mme_next()* to attempt playing another track in the track session, to create a new track session, to request input from the user, or to perform some other recovery task.

The MME_EVENT_FINISHED_WITH_ERROR event is delivered to indicate that the MME was unable to *initiate playback* of several consecutive tracks in a track session, and that, therefore, the tracks in the track session are probably not readable — they may be corrupt, in an unsupported format, etc. Once playback has started, read errors are handled by **`io-media`**. See the *MME Developer's Guide* chapter Playback Errors.

Event data

The ID of the track session, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`trksessions`**.

MME_EVENT_MEDIA_STATUS

The MME delivers the event MME_EVENT_MEDIA_STATUS to indicate a change in the status of media playback. Delivery of this event is triggered by a change to any of the following:

- title
- chapter
- angle
- subtitle
- audio

At present, this event is delivered only:

- for iPod devices
- when a chapter changes (**`mm_media_status_reason_t`** = MM_MEDIA_CHAPTER_UPDATE)

Event data

Media status information, and the reason for delivery of the event, in **`mm_media_status_event_t`**.

Database tables updated

No database tables are updated.

MME_EVENT_NEWOUTPUT

The MME delivers the event MME_EVENT_NEWOUTPUT when it has detected a new output device in the system. The device can then be attached to a zone. If the device is already part of a zone that is currently being used for playback, the MME will automatically attach the new device to that zone.

Event data

The ID of the new output device, in a **`uint64_t`**.

The following table is updated:

- **`outputdevices`**.

MME_EVENT_NOWPLAYING_METADATA

The MME delivers the event MME_EVENT_NOWPLAYING_METADATA when it is playing a track and has updated metadata in the **`nowplaying`** table. This behavior means that the MME delivers:

- One MME_EVENT_NOWPLAYING_METADATA event after every MME_EVENT_TRACKCHANGE event.
- A subsequent MME_EVENT_NOWPLAYING_METADATA event every time metadata is updated for the currently playing track.

  For example, if the MME is playing a track from a streamed source, it can start playback as soon as it has sufficient data to be able to continue playing without gaps, before the entire track and its metadata are downloaded. In this case, each time the MME receives new metadata for the track, it updates the **`nowplaying`** table, and delivers the MME_EVENT_NOWPLAYING_METADATA event to inform the client application of the update.

After receiving a MME_EVENT_NOWPLAYING_METADATA, the client application should query the **`nowplaying`** table for new metadata.

In most cases the client application will receive a MME_EVENT_NOWPLAYING_METADATA event immediately after receiving a MME_EVENT_TRACKCHANGE event. There are, however, two exceptions to this rule:

- If playback skips forward through tracks so quickly that any metadata that could be retrieved would no longer apply to the current track, the MME does *not* deliver a MME_EVENT_NOWPLAYING_METADATA event after each MME_EVENT_TRACKCHANGE event.
- If the new currently playing track is on a device that delays before making metadata available, the client application may experience a delay between MME_EVENT_TRACKCHANGE and MME_EVENT_NOWPLAYING_METADATA events. For more information, see "**`io-media`** option to set maximum wait for metadata" in the chapter Configuring Other Components of the *MME Configuration Guide*.

Event data

No data is delivered.

Database tables updated

The following table is updated:

- **`nowplaying`**

MME_EVENT_OUTPUTATTRCHANGE

The MME delivers the event MME_EVENT_OUTPUTATTRCHANGE when any output attribute changes.

Event data

The ID of the output where the change occured, in a **`uint64_t`**.

Database tables updated

No database tables are updated.

MME_EVENT_OUTPUTREMOVED

The MME delivers the event MME_EVENT_OUTPUTREMOVED when it detects that an output device has been removed. If the removed output device is the only active one on a zone where playback is underway, the MME will stop playback.

Event data

The ID of the removed output device, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`outputdevices`**

MME_EVENT_PLAYAUTOPAUSED

The MME delivers the event MME_EVENT_PLAYAUTOPAUSED after it has started playing a paused track. When it delivers this event, the MME is waiting for the client application to call the function *mme_play_set_speed()* with *speed* set to 1000 in order to begin playback of the track.

Event data

No data.

Database tables updated

No database tables are updated.

MME_EVENT_PLAY_ERROR

The MME delivers the event MME_EVENT_PLAY_ERROR when a playback error has occurred. Various playback errors trigger delivery of this event. The cause of the error is carried in the event data.

Event data

The cause of the error and the offset in **`mme_play_error_t`**.

Database tables updated

The database tables updated depends on the error that triggers delivery of the event.

MME_EVENT_PLAYLIST

For future use.

MME_EVENT_PLAYSTATE

The MME delivers the event MME_EVENT_PLAYSTATE when it changes the play state or the play speed in the control context. The play state is the type of playback underway in the control context. Examples of play states are playing, paused, seek to time, stopped, slow forward, fast forward, slow reverse and fast reverse.

This event is *not* delivered when playback changes tracks *unless* the play state or play speed have changed.

Event data

The current playstate and speed, in **`mme_playstate_speed_t`**.

Database tables updated

No database tables are updated.

MME_EVENT_PLAY_WARNING

The MME delivers the playback event MME_EVENT_PLAY_WARNING when **io-media** has indicated that it has detected a playback situation that requires a warning to the client application. The MME delivers this event if during playback of a file that is being ripped (or copied) **io-media**, due to fast forward or some other mechanism, playback advances so far that it risks over-running the end of the partially ripped file.

When the client application receives a MME_EVENT_PLAY_WARNING event, it should check the flags carried in the data structure **mm_warning_info_t** to determine the best course of action to take.

The first MME_EVENT_PLAY_WARNING event has **mm_warning_info_t.mm_warning** set to MM_WARNING_READ_TIMEOUT, indicating that playback is about to over-run the end of the ripped file, and that the client application can respond before playback reaches the end of the ripped file and the user hears a gap in the playback. Subsequent MME_EVENT_PLAY_WARNING events have **mm_warning_info_t.flags** set to MM_WARNING_FLAG_AUDIBLE, indicating that the user has encountered an audible gap in the playback.

Event data

mm_warning_info_t, with the appropriate warning flags set.

Database table updated

No database tables are updated.

MME_EVENT_RANDOMCHANGE

The MME delivers the event MME_EVENT_RANDOMCHANGE after it has changed the random play settings for a specified control context, or because a new track session has been set for the control context.

To find out the current random play mode for a control context, call the function *mme_getrandom()*. To change the random play mode of a control context, call the function *mme_setrandom()*. For more information about random mode settings, see *mme_setrandom()*.

The track number in a track session is determined differently in random playback and sequential playback modes. Therefore, if a client application receives the event MME_EVENT_RANDOMCHANGE it should call the function *mme_trksession_get_info()* to refresh its track number information.

Event data

The new random setting for the control context in **`mme_mode_random_t`**.

Database tables updated

The following tables are updated:

- **`track sessions`**
- **`controlcontexts`**

MME_EVENT_REPEATCHANGE

The MME delivers the event MME_EVENT_REPEATCHANGE after it has changed the repeat play settings for a specified control context, or because a new track session has been set for the control context. To find out the current repeat play mode for a control context, call the function *mme_getrepeat()*. To change the repeat play mode of a control context, call the function *mme_setrepeat()*. For more information about repeat mode settings, see *mme_setrepeat()*.

Event data

The new repeat setting for the control context, in **`mme_mode_repeat_t`**.

Database tables updated

The following tables are updated:

- **`track sessions`**
- **`controlcontexts`**

MME_EVENT_SCANMODECHANGE

The MME delivers the event MME_EVENT_SCANMODECHANGE after it has changed the scan mode for a specified control context. To find out the current scan mode for a control context, call the function *mme_getscanmode()* to get the new setting. To change the scan mode for a control context, call the function *mme_setscanmode()*.

Event data

No data.

Database tables updated

No database tables are updated.

MME_EVENT_TIME

A specified amount of time has passed during playback of a track.

Event data

A snapshot of current time information, in **`mme_time_t`**.

Database tables updated

No database tables are updated.

MME_EVENT_TRACKCHANGE

The MME delivers the event MME_EVENT_TRACKCHANGE when a track change occurs during playback.

Event data

This event returns the following data:

- the file ID (*fid*) of the track currently playing in **`mme_event_trackchange_t.fid`**
- the file ID (*fid_requested*) of the requested track in **`mme_event_trackchange_t.fid_requested`**.
- the offset (*offset*) of the requested track in **`mme_event_trackchange_t.offset`**.

The requested and playing file IDs may be different when playback is requested during a ripping operation.

Database tables updated

The following tables are updated:

- **`nowplaying`**
- **`track sessions`**

MME_EVENT_TRKSESSION

The MME delivers the event MME_EVENT_TRKSESSION when the track session in a control context has changed and a new track session ID (*trksessionid*) has been set for the control context, or when the number of tracks in a track session has changed.

Event data

The ID (*trksessionid*) of the new track session, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`controlcontexts`**

MME_EVENT_TRKSESSIONVIEW_COMPLETE

The MME delivers the event MME_EVENT_TRKSESSIONVIEW_COMPLETE when it has finished updating the track session view in the **`trksessinview`** table.

Event data

The ID of the track session, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`trksessionview`**

This event is delivered *only* when the MME has written to the **`tracksessionview`** table. For more information about configuring the MME track session view behavior, see "Configuring playback behavior" in the *MME Configuration Guide*.

MME_EVENT_TRKSESSIONVIEW_INVALID

The MME delivers the event MME_EVENT_TRKSESSIONVIEW_INVALID when a track session has changed, rendering the data in the **`trksessionview`** table invalid. This event indicates that MME will delete the data in the **`trksessionview`** table.

Event data

The ID of the track session whose information will be deleted from the **`trksessionview`** table, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`trksessionview`**

MME_EVENT_TRKSESSIONVIEW_UPDATE

The MME delivers the event MME_EVENT_TRKSESSIONVIEW_UPDATE when it has updated the data in the **`trksessionview`** table.

Event data

The ID of the track session whose data the MME has updated in the **`trksessionview`** table, in a **`uint64_t`**.

Database tables updated

The following table is updated:

- **`trksessionview`**

MME_EVENT_VIDEO_STATUS

The MME delivers MME_EVENT_VIDEO_STATUS after it has changed the status of a video it is playing, including:

- video resolution
- video aspect ratio

This information is in **`mm_event_data_t.video_status`**.

To find out the status of a video, call the function *mme_video_get_status()*. To change video attributes, call the relevant *mme_video_set_*()* function.

Event data

No data.

Database tables updated

No database tables are updated.

Playback error events

The MME playback error events are defined by the enumerated type **`mme_play_error_type_t`**:

```
typedef enum mme_play_error_type {
...
    MME_PLAY_ERROR_*
    ...
} mme_play_error_type_t;
```

Playback errors are grouped by the type of command that can produce the error. This definition is in the enumerated type **`mme_command_type_t`**:

```
typedef enum mme_command_type {
    MME_COMMAND_TYPE_PLAY    = 1,
    MME_COMMAND_TYPE_BUTTON  = 2
} mme_command_type_t;
```

The playback error events are:

- MME_PLAY_ERROR_BLOCKEDDOMAIN
- MME_PLAY_ERROR_BLOCKEDUOP
- MME_PLAY_ERROR_CORRUPT
- MME_PLAY_ERROR_DEVICEREMOVED
- MME_PLAY_ERROR_INPUTUNDERRUN
- MME_PLAY_ERROR_INVALIDFID

- MME_PLAY_ERROR_INVALIDSAVEDSTATE
- MME_PLAY_ERROR_MEDIABUSY
- MME_PLAY_ERROR_NETWORK
- MME_PLAY_ERROR_NOEXIST
- MME_PLAY_ERROR_NOOUTPUTDEVICES
- MME_PLAY_ERROR_NORIGHTS
- MME_PLAY_ERROR_NOTSPECIFIED
- MME_PLAY_ERROR_OUTPUTFAILEDTOATTACH
- MME_PLAY_ERROR_OUTPUTUNDERRUN
- MME_PLAY_ERROR_PARENTALCONTROL
- MME_PLAY_ERROR_READ
- MME_PLAY_ERROR_REGION
- MME_PLAY_ERROR_UNSUPPORTEDCODEC

MME_PLAY_ERROR_BLOCKEDDOMAIN

The MME delivers the event MME_PLAY_ERROR_BLOCKEDDOMAIN when it blocks a user operation that is forbidden by a domain mask on the media source. For more information about these domains, see *mme_dvd_get_status()*.

Event data

The type of user operation that was attempted, in **`mme_play_error_t.command_error.command`**, and:

- the file ID (*fid*): **`mme_play_error_t.command_error.fid`**, for play commands (MME_COMMAND_TYPE_PLAY)
- the button: **`mme_play_error_t.command_error.button`**, for button commands (MME_COMMAND_TYPE_BUTTON).

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_BLOCKEDUOP

The MME delivers the event MME_PLAY_ERROR_BLOCKEDUOP when it blocks a user operation. This sort of situation can occur when, for example, the user attempts to use a forbidden play or button command.

Event data

The type of user operation that was attempted, in **`mme_play_error_t.command_error.command`**, and:

- for play commands (MME_COMMAND_TYPE_PLAY), the file ID (*fid*), in **`mme_play_error_t.command_error.fid`**
- for button commands (MME_COMMAND_TYPE_BUTTON), the button, in **`mme_play_error_t.command_error.button`**

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_CORRUPT

The MME delivers the event MME_PLAY_ERROR_CORRUPT when the MME is unable to play a file for which it has the correct codec.

Event data

The file ID (*fid*) of the corrupt file, in **`mme_play_error_t.fid`**.

Database tables updated

The following table is updated:

- **`library`**. The *playable* field for the file is set to 0.

MME_PLAY_ERROR_DEVICEREMOVED

The MME delivers the event MME_PLAY_ERROR_DEVICEREMOVED when the mediastore (or device and mediastore) from which it is playing is removed from the system. The MME will stop playback, deliver this error event, then deliver the event MME_EVENT_PLAY_ERROR.

Event data

The mediastore ID (*msid*) of the mediastore with the file or track that was being played when the mediastore was removed, in **`mme_play_error_t.msid`**.

Database tables updated

The following table is updated:

- **`mediastores`**

MME_PLAY_ERROR_INPUTUNDERRUN

The MME delivers the event MME_PLAY_ERROR_INPUTUNDERRUN when it encounters problems filling its input buffer and has an input underrun. An input underrun is usually caused by slow input media. The

MME_PLAY_ERROR_INPUTUNDERRUN signals a warning: the input underrun results in an audible gap during playback, but playback will continue.

Event data

The file ID (*fid*) of the file that had the input underrun, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_INVALIDFID

The MME delivers the event MME_PLAY_ERROR_INVALIDFID when it is requested to play an invalid file ID. This situation can occur if the requested *fid* is not found in the **`library`** table, or if it is not included in the currently active track session.

Event data

The invalid *fid* number, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_MEDIABUSY

The MME delivers the event MME_PLAY_ERROR_MEDIABUSY when it has attempted to start playback of a file, but the mediastore is being used by another operation and is locked.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_INVALIDSAVEDSTATE

The MME delivers the event MME_PLAY_ERROR_INVALIDSAVEDSTATE if it attempts but fails to resume playback on a mediastore. Delivery of this event indicates that the data saved for the track session is incorrect or corrupt.

Event data

The file ID of the track where the error occurred, in **`mme_play_error_t.fid`**.

Database table updated

No database tables are updated.

MME_PLAY_ERROR_NETWORK

The MME delivers the event MME_PLAY_ERROR_NETWORK when a network error had caused playback to fail.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_NOEXIST

The MME delivers the event MME_PLAY_ERROR_NOEXIST when it has failed to play a file because the file does not exist, or because the file's specified content was not found. This situation can occur in situations such as the following:

- A file is removed from a mediastore and the MME receives a request to play the file before it has performed the first pass of a resynchronization on the mediastore. Because the mediastore was not resynchronized, the MME could not know that the file had been removed until it attempted to play the file.
- A file is corrupt and the MME cannot read its content.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_NOOUTPUTDEVICES

The MME delivers the event MME_PLAY_ERROR_NOOUTPUTDEVICES when it starts playback of a file, but no output devices are attached to the control context with the playback.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_NORIGHTS

The MME delivers the event MME_PLAY_ERROR_NORIGHTS when it encounters a DRM protected file which it is not licensed to decrypt and cannot play.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_NOTSPECIFIED

The MME delivers the event MME_PLAY_ERROR_NOTSPECIFIED when playback fails for a reason not covered by the other playback error events, or when the MME is unable to determine the cause of the failure.

Event data

No event data is delivered.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_OUTPUTFAILEDATTACH

The MME delivers the event MME_PLAY_ERROR_OUTPUTFAILEDATTACH when it has failed to attach an output that is part of its current zone. The MME_PLAY_ERROR_OUTPUTFAILEDATTACH signals a warning: playback continues.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_PARENTALCONTROL

The MME delivers the event MME_PLAY_ERROR_PARENTALCONTROL when it parental control settings have prevented it from playing a requested track or file.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_READ

The MME delivers the event MME_PLAY_ERROR_READ when it is playing a track or file and it encounters a read error.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_REGION

The MME delivers the event MME_PLAY_ERROR_REGION when it is unable to continue playback because the regional settings for a mediastore do not correspond to the regional settings for the device playing the mediastore.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_OUTPUTUNDERRUN

The MME delivers the event MME_PLAY_ERROR_OUTPUTUNDERRUN when its output buffer is drained during playback and the MME has encountered an output underrun. An output underrun is usually caused by a slow system decode process, which is unable to keep up with audio output at normal speed (1 ×). The MME_PLAY_ERROR_OUTPUTUNDERRUN signals a warning: the output underrun results in an audible gap during playback, but playback will continue.

Event data

The file ID (*fid*) of the file that had the output underrun, in **`mme_play_error_t.fid`**.

Database tables updated

No database tables are updated.

MME_PLAY_ERROR_UNSUPPORTEDCODEC

The MME delivers the event MME_PLAY_ERROR_UNSUPPORTEDCODEC after it starts playback and determines that it does not have the codec it needs to decode the stream it is attempting to play.

Event data

The file ID (*fid*) of the file that failed playback, in **`mme_play_error_t.fid`**.

Database tables updated

The following table is updated:

- **`library`**. The *playable* field for the file is set to 0.

Chapter 5

MME Media Copy and Ripping Events

In this chapter. . .

MME events are like other QNX Neutrino events. They are signals or pulses used to notify a client application thread that a particular condition has occurred. Unlike signals and pulses, events can be used to carry data.

This chapter includes:

- Media copy and ripping events
- Media copy and ripping error events

For other information about other types of MME events, see the following chapters in this reference:

- MME Events
- MME Synchronization Events
- MME Playback Events
- MME Metadata Events

For more information about events in general, see the *QNX Neutrino Programmer's Guide*.

Media copying and ripping events

The MME delivers media copying and ripping events (MME_EVENT_CLASS_COPY) to the client application to indicate the status or result of a media copying or ripping activity.

The MME media copying and ripping events are:

- MME_EVENT_COPY_ERROR
- MME_EVENT_MEDIACOPIER_COPYFID
- MME_EVENT_MEDIACOPIER_SKIPFID
- MME_EVENT_MEDIACOPIER_STARTFID
- MME_EVENT_MEDIACOPIER_COMPLETE
- MME_EVENT_MEDIACOPIER_DISABLED

MME_EVENT_COPY_ERROR

The MME delivers the event MME_EVENT_COPY_ERROR when it encounters a copying or ripping error. The cause of the error is carried in the event data.

Event data

The copy error type **`mme_copy_error_type_t`**, in **`mme_copy_error_t`**.

Database tables updated

The MME updates different tables, depending on the error.

MME_EVENT_MEDIACOPIER_COPYFID

The MME delivers the event MME_EVENT_MEDIACOPIER_COPYFID when it has finished copying or ripping a file.

Event data

The structure **`mme_copy_info_t`** with the IDs of:

- the source file that was copied (*srcfid*)
- the destination file that was created by the copy (*dstfid*)
- the entry in the **`copyqueue`** table (*cqid*) for the file that was copied

Database tables updated

The following tables are updated:

- **`nowplaying`**
- **`library_*`** (**`library_artists`**, **`library_genres`**, etc.)
- **`copyqueue`**

MME_EVENT_MEDIACOPIER_SKIPFID

The MME delivers the event MME_EVENT_MEDIACOPIER_SKIPFID when it skips copying or ripping of a specified file (*fid*) because the mediacopier is disabled, or because playback has priority access to the media source with the file.

Event data

The structure **`mme_copy_info_t`** with the IDs of:

- the source file that was skipped (*srcfid*)
- the destination file was to be created by the copy (*dstfid*)
- the entry in the **`copyqueue`** table (*cqid*) that was skipped

Database tables updated

The following table is updated:

- **`copyqueue`**

MME_EVENT_MEDIACOPIER_STARTFID

The MME delivers the event MME_EVENT_MEDIACOPIER_STARTFID when it starts a file copying or ripping operation.

Event data

The structure **`mme_copy_info_t`** with the IDs of:

- the source file that will be copied (*srcfid*)
- the destination file that will be created by the copy (*dstfid*)
- the entry in the **`copyqueue`** table (*cqid*) for the file that will be copied

Database tables updated

The following table is updated:

- **`copyqueue`**

MME_EVENT_MEDIACOPIER_COMPLETE

The MME delivers the event MME_EVENT_MEDIACOPIER_COMPLETE when it has finished copying or ripping all files listed in the copy queue: the copy queue is empty.

Event data

No event data is delivered.

Database tables updated

The following table is updated:

- **`copyqueue`**.

MME_EVENT_MEDIACOPIER_DISABLED

The MME delivers the event MME_EVENT_MEDIACOPIER_DISABLED when its media copying and ripping capabilities have been disabled.

Event data

No event data is delivered.

Database tables updated

The following table is updated:

- **`copyqueue`**

Media copying and ripping error events

The MME media copying and ripping error events are defined by the enumerated type **mme_copy_error_type_t**:

```
typedef enum mme_copy_error_type {
    ...
    MME_COPY_ERROR_*
    ...
} mme_copy_error_type_t;
```

The MME media copying and ripping error events are:

- MME_COPY_ERROR_CORRUPTION
- MME_COPY_ERROR_DEVICEREMOVED
- MME_EVENT_COPY_FATAL_ERROR
- MME_COPY_ERROR_FILEEXISTS
- MME_COPY_ERROR_MEDIABUSY
- MME_COPY_ERROR_MEDIAFULL
- MME_COPY_ERROR_NORIGHTS
- MME_COPY_ERROR_NOTSPECIFIED
- MME_COPY_ERROR_READ
- MME_COPY_ERROR_WRITE

MME_COPY_ERROR_CORRUPTION

The MME delivers the event MME_COPY_ERROR_CORRUPTION when it has attempted to copy or rip a file from a mediastore and failed because the file is corrupt. When the MME media copy process encounters this condition, it advances to the next entry in the copy queue and attempts to copy or rip the file for that entry.

If the **<DeleteOnNonRecoverableError>** MME configuration option is enabled, the MME deletes the copy queue entry for the corrupt file and delivers the MME_EVENT_COPY_FATAL_ERROR event.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **mme_event_copy_error_t.cqid**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_DEVICEREMOVED

The MME delivers the event MME_COPY_ERROR_DEVICEREMOVED when it has attempted to copy or rip a file from a mediastore or to a mediastore that has been removed from the system. When the MME media copy process encounters this condition, it moves to the next entry in the copy queue and attempts to copy or rip that file.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_EVENT_COPY_FATAL_ERROR

The MME delivers the event MME_EVENT_COPY_FATAL_ERROR when it has deleted an entry from the copy queue table because it has determined that the item cannot be copied or ripped.

This event is only delivered if the **`<DeleteOnNonRecoverableError>`** MME configuration option is enabled, configuring the MME to delete from the copy queue files that cannot be copied or ripped.

Event data

The code for the error that caused the copy or ripping to fail, in **`mme_copy_error_t`**.

Database tables updated

Tables are updated according to the error that causes the event to be delivered.

MME_COPY_ERROR_FILEEXISTS

The MME delivers the event MME_COPY_ERROR_FILEEXISTS when it has attempted to copy or rip a file that already exists in the destination mediastore. When the MME media copy process encounters a file that already exists in the destination mediastore, depending on the overwrite setting configured by the configuration element **`<FileOverwrite>`**, it either skips the requested file and moves to the next entry in the copy queue, or it overwrites the file.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_MEDIABUSY

The MME delivers the event MME_COPY_ERROR_MEDIABUSY when it has attempted to copy or rip from a mediastore that was already in use. When the MME media copy process encounters a busy mediastore, it skips the requested file and moves to the next entry in the copy queue.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_MEDIAFULL

The MME delivers the event MME_COPY_ERROR_MEDIAFULL when the destination mediastore does not have enough space to complete the requested media copying or ripping operation. The MME may deliver this event before starting a media copying or ripping operation, or during the operation.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_NORIGHTS

The MME delivers the event MME_COPY_ERROR_NORIGHTS when it has attempted to copy or rip a file from a mediastore and failed because the source file is DRM protected and the system is not licensed to copy it. When the MME media copy process encounters this condition, it advances to the next entry in the copy queue and attempts to copy or rip the file for that entry.

If the **`<DeleteOnNonRecoverableError>`** MME configuration option is enabled, the MME deletes the copy queue entry for the protected file and delivers the MME_EVENT_COPY_FATAL_ERROR event.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_NOTSPECIFIED

The MME delivers the event MME_COPY_ERROR_NOTSPECIFIED when a media copying or ripping operation fails for a reason not covered by the other media copying and ripping error events, or when the MME is unable to determine the cause of the failure.

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_READ

The MME delivers the event MME_COPY_ERROR_READ when it is copying or ripping a file and it encounters a read error.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

MME_COPY_ERROR_WRITE

The MME delivers the event MME_COPY_ERROR_WRITE when it is copying or ripping a file and it encounters a write error.

Event data

The ID of the copy queue entry (*cqid*) of the file that could not be copied or ripped, in **`mme_copy_error_t.cqid`**.

Database tables updated

No database tables are updated.

Chapter 6

MME Metadata Events

In this chapter. . .

MME events are like other QNX Neutrino events. They are signals or pulses used to notify a client application thread that a particular condition has occurred. Unlike signals and pulses, events can be used to carry data.

This chapter includes:

- Metadata events

For other information about other types of MME events, see the following chapters in this reference:

- MME Events
- MME Synchronization Events
- MME Playback Events
- MME Media Copy and Ripping Events

For more information about events in general, see the *QNX Neutrino Programmer's Guide*.

Metadata events

The MME delivers metadata events (MME_EVENT_CLASS_METADATA) to the client application to indicate the status or result of a metadata retrieval operation, and for successful operations, the metadata request ID.

The MME metadata events are:

- MME_EVENT_METADATA_IMAGE
- MME_EVENT_METADATA_INFO

MME_EVENT_METADATA_IMAGE

The MME delivers the event MME_EVENT_METADATA_IMAGE after the client application calls *mme_metadata_image_load()*:

- to deliver the metadata request ID to the client application
- when the call to *mme_metadata_image_load()* is asynchronous, to indicate that the function has completed its task

Event data

The metadata structure type **`mme_event_metatadata_image_t`**, with:

- the metadata request ID, as a **`uint64_t`**
- an error number, as a **`int`**; set to EOK if the call to the function is successful
- the URL location of the requested image, in the structure **`mme_metadata_image_url_t`**

Database tables updated

No database tables are updated.

MME_EVENT_METADATA_INFO

The MME delivers the event MME_EVENT_METADATA_INFO after the client application calls one of the *mme_metadata_getinfo_*()* functions:

- to deliver the metadata request ID to the client application
- when the call to the **<***mme_metadata_getinfo_*()* function is asynchronous, to indicate that the function has completed its task

Event data

The metadata structure type **`mme_event_metatadata_info_t`**, with:

- the metadata request ID, as a **`uint64_t`**
- an error number, as a **`int`**; set to EOK if the call to the function is successful
- a **`mme_metadata_info_t`** that includes A NULL-terminated XML formated string containing metadata.

Database tables updated

No database tables are updated.

Appendix A

MME Database Schema Reference

In this appendix. . .

This is a reference of the tables and indexes in the MME database.

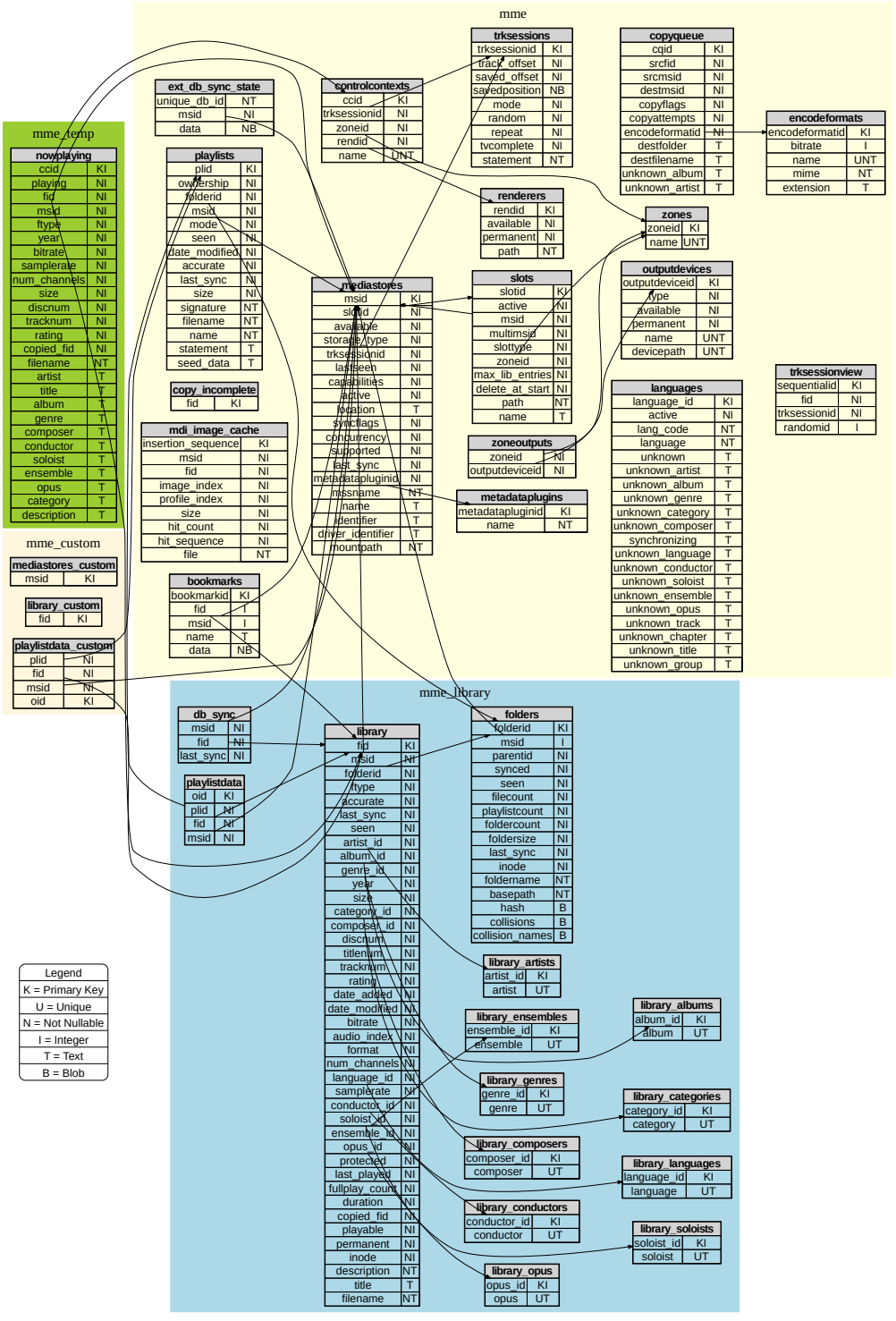
Tables that begin with an underscore character (_) are not documented here. They are for internal use only.

Tables:

- controlcontexts
- renderers
- zones
- zoneoutputs
- outputdevices
- slots
- languages
- mediastores
- metadataplugins
- playlists
- trksessions
- encodeformats
- copyqueue
- bookmarks
- trksessionview
- copy_incomplete
- mdi_image_cache
- ext_db_sync_state
- folders
- library
- library_genres
- library_artists
- library_albums
- library_composers

- library_conductors
- library_soloists
- library_ensembles
- library_opus
- library_categories
- library_languages
- db_sync
- playlistdata
- nowplaying
- mediastores_custom
- library_custom
- playlistdata_custom

Table diagrams:



The MME Schema.

Tables in mme

Table: controlcontexts

The **controlcontexts** table defines MME control contexts. Control contexts define where clients can connect to the MME and control it. Each control context can play one media track at a time, manage its own list of items to play, and output playback to one zone. Control contexts are statically configured and enumerated at startup time.

Primary key: ccid

Fields:

Field	Description	Type	Default	Nulls?	References
ccid	The control context ID.	Integer		yes	
trksessionid	The ID of the tracksession that is being used on the control context.	Integer	0	no	trksessions
zoneid	The ID of the zone to which the control context is outputting playback.	Integer	0	no	zones
rendid	The ID of the renderer that this control context should use.	Integer		no	renderers
name	The name of the control context. This name will appear as /dev/mme/*name*	Text, unique		no	

Table: renderers

The **renderers** table defines the **io-media** instances that exist in the system, and the capabilities of these **io-media** instances. A control context uses the specified **io-media** to decoding and encode work.

Primary key: rendid

Fields:

Field	Description	Type	Default	Nulls?	References
rendid	The ID of the renderer instance.	Integer		yes	
available	The renderer availability. Set to 1 if this renderer can be used, 0 if it can't be used.	Integer	1	no	

continued...

Field	Description	Type	Default	Nulls?	References
permanent	Permanent renderers may not be removed.	Integer	1	no	
path	The path to the renderer. For example: **/net/***node***/dev/io-media**.	Text		no	

Table: zones

The **zones** table defines the MME zones. The output devices associated with a zone are listed in the **zoneoutputs** table.

Primary key: zoneid

Fields:

Field	Description	Type	Default	Nulls?	References
zoneid	The ID of the zone	Integer		yes	
name	The name of the zone	Text, unique		no	

Table: zoneoutputs

The **zoneoutputs** table lists the output devices that are associated with each zone. Each *zoneid* can have multiple rows to support multiple output devices.

No primary key.

Fields:

Field	Description	Type	Default	Nulls?	References
zoneid	The ID of the zone.	Integer		no	zones
outputdeviceid	The IDs of the output devices in the zone.	Integer		no	outputdevices

Table: outputdevices

The **outputdevices** table lists known output devices. Output devices define where media can be sent. An output device could be a GF layer, an **io-audio** PCM name, a Bluetooth headset, etc.

Primary key: outputdeviceid

Fields:

Field	Description	Type	Default	Nulls?	References
outputdeviceid	The ID of the output device.	Integer		yes	
type	The type of device, as defined by the enumerated type **mme_outputtype_t** values: OUTPUTTYPE_*.	Integer	0	no	
available	The availability of the output device. Set to 1 for available.	Integer	1	no	
permanent	The device permanency. Set to 1 to make the device permanent and forbid its removal.	Integer		no	
name	The name of the device. This name can be shared with end users.	Text, unique		no	
devicepath	The location of the output device, used to connect to the output device. This path is not shared with end users.	Text, unique		no	

Table: slots

The **slots** table lists the slots known to the MME. Slots define the physical locations where the MME looks for new mediastores. The default setup assumes two USB mass storage devices, one CD/DVD drive, and the hard drive. You may wish to customize where the location of the hard drive. In addition, if you add control contexts and they have their own slots, you must add them to this table. Note that the local control context's hard drive must be the first entry in the table, with *msid* = 1.

Primary key: slotid

Fields:

Field	Description	Type	Default	Nulls?	References
slotid	The ID for the slot.	Integer		yes	
active	Indicates whether the slot is active (available), or unavailable: • 1 = active	Integer	0	no	
msid	The ID of the mediastore associated with this slot.	Integer	0	no	mediastores

continued...

Field	Description	Type	Default	Nulls?	References
multimsid		Integer	0	no	
slottype	The type of slot. These correspond to the MME_SLOTTYPE_* types defined in **<mme/interface.h>**: • 0 = standard • 1 = USB • 2 = CD/DVD • 3 = harddrive • 4 = media file system (**io-fs**)	Integer	0	no	
zoneid	The ID of the zone associated with this slot.	Integer		no	zones
max_lib_entries	The maximum number of library table entries an active media store in this slot is permitted to use. A value of 0 means there is no limit enforced. @delete_at_start If non-zero, mediastores that were listed as active at shutdown in this slot are deleted instead of being set to unvavailable.	Integer	0	no	
delete_at_start		Integer	0	no	
path	The filesystem path to this slot.	Text		no	
name	The slot name. This name is used as the default for mediastores without names.	Text	NULL	yes	

Indices:

Index name	Fields
slots_msid_index	msid

Table: languages

The **languages** table defines strings that your application can use for multi-language support.

Primary key: language_id

Fields:

Field	Description	Type	Default	Nulls?	References
language_id	The language ID.	Integer		yes	
active	Indicates whether this is the active (current) language. • 1 = active	Integer	0	no	
lang_code	The 2-character ISO639-1 language code.	Text		no	
language	The language name.	Text		no	
unknown	String for "unknown".	Text		yes	
unknown_artist	String for "unknown artist"	Text		yes	
unknown_album	String for "unknown album"	Text		yes	
unknown_genre	String for "unknown genre"	Text		yes	
unknown_category	String for "unknown category"	Text		yes	
unknown_composer	String for "unknown composer"	Text		yes	
synchronizing	String for "synchronizing"	Text		yes	
unknown_language		Text		yes	
unknown_conductor		Text		yes	
unknown_soloist		Text		yes	
unknown_ensemble		Text		yes	
unknown_opus		Text		yes	
unknown_track	String for building unknown title of CDDA and DVD-Audio tracks	Text	NULL	yes	
unknown_chapter	String for building unknown title of DVD-Video tracks	Text	NULL	yes	
unknown_title	String for building unknown title of DVD-Video tracks	Text	NULL	yes	
unknown_group	String for building unknown title of DVD-Audio tracks	Text	NULL	yes	

Table: mediastores

The **mediastores** table lists the mediastores known to the MME. A mediastore is a collection of media tracks and/or files that the MME can access and play. Where a slot is the physical location of some media (for example, a CDROM drive), a mediastore represents the media itself (for example, a CD).

Mediastores are managed by the MME, so you don't need to customize this table.

Primary key: msid

Fields:

Field	Description	Type	Default	Nulls?	References
msid	The mediastore ID.	Integer		yes	
slotid	The ID of the physical slot associated with this mediastore.	Integer	0	no	slots
available	Indicates whether the mediastore is available: • 0 = not available, • 1 = available	Integer	0	no	
storage_type	The storage type, which corresponds to the MME_STORAGETYPE_* types defined in **<mme/interface.h>**.	Integer	0	no	
trksessionid	The last tracksession that was saved on this mediastore.	Integer	0	no	trksessions
lastseen	The last time the mediastore was seen by the MME. If there is no RTC in the system, this value will increment each time the mediastore is seen, but it will not show the true time.	Integer	0	no	
capabilities	The capabilities of this mediastore (for example, can it be explored or synchronized?). These capabilities correspond to the MME_MSCAP_* type defined in **<mme/interface.h>**.	Integer	0	no	
active	Indicates if a mediastore is active, or if a slot change is required. A mediastore can not be active if it is not available: • 0 = not active, • 1 = active (i.e. currently active slot in a changer)	Integer	0	no	
location	The location of the device where the mediastore is currently inserted: • empty = not currently inserted in a slot, or is in a device for which location has no meaning • non-empty = location string that has meaning only to the device (devices that only support one location will always be set to empty)	Text		yes	

continued...

Field	Description	Type	Default	Nulls?	References
syncflags	Indicates which synchronizations have been completed on the mediastore: • 0 = none (not synchronized) • 1 = pass 1 (files) • 2 = pass 2 (metadata) • 4 = pass 3 (playlists) • others to be determined	Integer	0	no	
concurrency	Indicates how many concurrent readers are supported by the mediastore: 1="one reader", 2="two readers", etc.	Integer	1	no	
supported	Indicates if the device is supported: • 0 = not supported • 1 = supported	Integer	1	no	
last_sync	The time (in nanoseconds from the reference) of the last synchronization attempt of any time on the mediastore.	Integer	0	no	
metadatapluginid	The metadataplugin that was used to sync the mediastore. 0 means not specified.	Integer	0	no	metadataplugins
mssname	Internal use only. The MSS plugin that handles this mediastore.	Text		no	
name	The name of the mediastore (for example, "memory stick". This field may be null if the mediastore name cannot be determined.	Text	NULL	yes	
identifier	A unique identifier, such as the FAT serial number. Set to NULL to flag the mediastore as invalid and ready to be deleted in the background.	Text		yes	
driver_identifier	A unique identifier, as provided by the device driver.	Text		yes	
mountpath	The mounted path of the mediastore.	Text		no	

Indices:

Index name	Fields
mediastores_identifier_index	identifier
mediastores_driver_identifier_index	driver_identifier

continued...

Index name	Fields
mediastores_active_index	active

Table: metadataplugins

The **metadataplugins** table lists the metadata syncronizers known to the MME.

Primary key: metadatapluginid

Fields:

Field	Description	Type	Default	Nulls?	References
metadatapluginid	The metadata plugin ID.	Integer		yes	
name	The name of the metadata plugin.	Text		no	

Table: playlists

The **playlists** table holds playlists that your application can convert into track sessions and play. A playlist is a collection of media tracks. Each playlist is defined by an SQL statement that queries the library for tracks that meet some criteria. Alternately, the SQL statement may query the **playlistdata** table, which can contain an arbitrary selection of tracks, grouped by a matching playlist ID.

Primary key: plid

Fields:

Field	Description	Type	Default	Nulls?	References
plid	The playlist ID.	Integer		yes	
ownership	Indicates who owns this playlist: • 0 = owned by the MME • 1 = owned by the device • 2 = owned by the user	Integer	0	no	
folderid	The ID of the folder that the playlist is in.	Integer	0	no	folders
msid	A link to a mediastore. If this playlist belongs to more than one mediastore, then this msid is 0.	Integer	0	no	mediastores

continued...

Field	Description	Type	Default	Nulls?	References
mode	The playlist mode: • 0 = library mode • 1 = generated mode	Integer	0	no	
seen	Indicates that the file was seen during the latest synchronization. This field is set to 0 at the beginning of a synchronization, then set to 1 when the file is found.	Integer	1	no	
date_modified	The date this playlist was last modified.	Integer	0	no	
accurate	If this field is set to 1, the playlist is accurate.	Integer	0	no	
last_sync	The time (in nanoseconds from the reference) of the last playlist (pass 3) synchronization attempt for the playlist.	Integer	0	no	
size	The size of the playlist file on the device.	Integer	0	no	
signature	md5 hash of the playlist.	Text	'0'	no	
filename	If the playlist points to a device, the filename of the playlist on the device. This name is a path relative to the basepath of the folder.	Text	"	no	
name	The playlist name.	Text		no	
statement	An SQL statement that returns a list of file IDs (*fid*s), either from the **`library`** table, or from the **`playlistdata`** table.	Text		yes	
seed_data	Used by playlist generators (i.e. mode = 1)	Text		yes	

Table: trksessions

The **`trksessions`** table stores track sessions, which are lists of file IDs(*fid*s) that the MME can access and play. A track session can be generated by using a playlist, or by any query to the **`library`** table that results in a list of file IDs (selecting all tracks by an artist, for example).

CAUTION: Your application shouldn't write to this table directly. It can create track sessions by calling the *mme_newtrksession()* function.

The fields in the **`trksessions`** table should only be accessed through MME function calls. The MME may cache some of the values in this table, so if the client application reads this table directly it may have incorrect data.

Primary key: trksessionid

Fields:

Field	Description	Type	Default	Nulls?	References
trksessionid		Integer		yes	
track_offset	Internal use only.	Integer	0	no	
saved_offset	The saved fid used to resume the trksession (0 = not saved).	Integer	0	no	
savedposition	The saved position in a *fid*/*bid* that can be used for resuming playback.	BLOB	0	no	
mode	The track session mode: • 0 = library mode	Integer	0	no	
random		Integer	0	no	
repeat		Integer	0	no	
tvcomplete	Indicates if the track view was finished loading: • 0 = no • 1 = yes	Integer	0	no	
statement	The SQL statement that results in a list of file IDs that the track session plays.	Text		no	

Table: encodeformats

The **`encodeformats`** table defines encode formats that can be used by the MME. Note that codecs that support multiple mime types or multiple bitrates will have separate entries in this table.

Primary key: encodeformatid

Fields:

Field	Description	Type	Default	Nulls?	References
encodeformatid	The endcode format ID.	Integer		yes	
bitrate	The bitrate to encode at, in kilobytes.	Integer	0	yes	
name	The name for the encode format.	Text, unique		no	
mime	The mime type to use.	Text		no	
extension	The output file extension.	Text		yes	

Table: copyqueue

The **copyqueue** table is a queue of files to copy from one mediastore to another. While the files are being copied, they may also be encoded ("ripped"). If the files are encoded, the encode format is defined by the **encodeformats** table.
Primary key: cqid

Fields:

Field	Description	Type	Default	Nulls?	References
cqid	Copy queue ID.	Integer		yes	
srcfid	The ID of the source file to copy.	Integer		no	
srcmsid	The ID of the source mediastore.	Integer		no	
destmsid	The ID of the destination mediastore.	Integer		no	
copyflags	Copy flags supplied by user	Integer	0	no	
copyattempts	The number of failed copy attempts to make before removing the item from the copy queue.	Integer	0	no	
encodeformatid	The encode format to use for the copy.	Integer	1	no	encodeformats
destfolder	The destination folder basepath name, in the format **/xxxxxx/**.	Text		yes	
destfilename	The destination filename. Don't add the extension. If this field is NULL, the MME will create a name.	Text		yes	
unknown_album	Metadata used to replace unknown album (if nonaccurate)	Text		yes	
unknown_artist	Metadata used to replace unknown artist (if nonaccurate)	Text		yes	

Table: bookmarks

The **bookmarks** table contains information about all bookmarks for file IDs.

Primary key: bookmarkid

Fields:

Field	Description	Type	Default	Nulls?	References
bookmarkid	The bookmark ID.	Integer		yes	
fid	The file ID of the bookmarked track.	Integer		yes	library
msid	The mediastore ID for the mediastore with the bookmarked file.	Integer		yes	mediastores
name	A name for the bookmark, specified with *mme_bookmark_create()*.	Text		yes	
data	Data used for resuming playback at the proper location. Internal use only.	BLOB		no	

Indices:

Index name	Fields
bookmarks_index_fid	fid
bookmarks_index_msid	msid

Table: trksessionview

The **`trksessionview`** table contains a snapshot of the current track session. All its fields are updated by the functions *mme_settrksession()* and *mme_trksessionview_update()*.

Primary key: sequentialid

Fields:

Field	Description	Type	Default	Nulls?	References
sequentialid	The track file IDs (*fid*) in sequential order, based on the results of the ORDER BY clauses in the SQL statement used to create the track session.	Integer		yes	
fid	The file ID of the track.	Integer		no	
trksessionid	The track session ID.	Integer		no	

continued...

Field	Description	Type	Default	Nulls?	References
randomid	The track file IDs (*fid*), in pseudo-random order. If random mode is turned on for the control context, the MME will play tracks in the order they appear in this field.	Integer		yes	

Indices:

Index name	Fields
trksessionview_index_random	trksessionid, randomid
trksessionview_index_seq	trksessionid, sequentialid

Table: copy_incomplete

Primary key: fid

Fields:

Field	Description	Type	Default	Nulls?	References
fid		Integer		yes	

Table: mdi_image_cache

The **imagecache** table contains a list of all image files stored in the persistent cache.

Primary key: insertion_sequence

Fields:

Field	Description	Type	Default	Nulls?	References
insertion_sequence	An id that keeps track of insertion order.	Integer		yes	
msid	The MSID the source image file was from.	Integer		no	
fid	The file ID of the track.	Integer		no	

continued...

Field	Description	Type	Default	Nulls?	References
image_index	The image index of a given track	Integer		no	
profile_index	The profile index for a converted image. -1 not converted.	Integer		no	
size	The size in bytes of the given index.	Integer		no	
hit_count	The number of cache hits for this entry.	Integer		no	
hit_sequence	A sequence number that indicates the last hit file.	Integer		no	
file	The relative path to the file within the cache.	Text		no	

Table: ext_db_sync_state

The **ext_db_sync_state** table contains persistent state information for all extern DBs that must stay in sync with the MME database.

No primary key.

Fields:

Field	Description	Type	Default	Nulls?	References
unique_db_id		Text		no	
msid	The MSID that corresponds to the state.	Integer		no	mediastores
data	The state data of the external DB	BLOB		no	

Tables in mme_library

Table: folders

The **folders** table stores the path of files found on mediastores and can be used to hierarchically find folders.

Primary key: folderid

Fields:

Field	Description	Type	Default	Nulls?	References
folderid	The folder ID for the folder.	Integer		yes	
msid	The mediastore to which the folder belongs.	Integer		yes	mediastores
parentid	The parent folder for this folder. Set to 0 if there is no parent folder.	Integer	0	no	
synced	If this field is set to 1, the folder has been synchronized during the first synchronization pass.	Integer	0	no	
seen	A flag to indicate if the folder was seen or not seen during synchronization.	Integer	1	no	
filecount	The number of files in the folder.	Integer	0	no	
playlistcount	The number of playlists in the folder.	Integer	0	no	
foldercount	The number of subfolders in the folder.	Integer	0	no	
foldersize	The size of the folder, in bytes.	Integer	0	no	
last_sync	Reserved for the time (in nanoseconds from the reference) of the last synchronization attempt on the mediastore.	Integer	0	no	
inode	Optional. The inode for the associated file.	Integer	0	no	
foldername	The name of the folder (for example, **`Rolling Stones`**).	Text		no	
basepath	The full path of the folder (for example, **`Music/Rolling Stones`**).	Text		no	
hash	For internal use only.	BLOB		yes	
collisions	For internal use only.	BLOB		yes	
collision_names	For internal use only.	BLOB		yes	

Table: library

The **`library`** table defines the media library used by the MME. Each entry in this table is a media track, which you can use to build track sessions and playlists.

The **`library`** is managed by the MME, so you don't need to customize it.

Primary key: fid

Fields:

Field	Description	Type	Default	Nulls?	References
fid	The file ID (*fid*) for the media track.	Integer		yes	
msid	The mediastore that this track is stored on.	Integer	0	no	mediastores
folderid	The path in the mediastore where the track is located.	Integer	0	no	folders
ftype	The type of the media track, which corresponds to the FTYPE_* types defined in **<mme/meinterface.h>**: • 0 = unknown • 1 = audio • 2 = video • 3 = audio and video • 4 = photo	Integer	0	no	
accurate	Indicates if the metadata for the track is known to be accurate.	Integer	0	no	
last_sync	The time (in nanoseconds from the reference) of the last metadata synchronization attempt for the track.	Integer	0	no	
seen	Indicates that the track has been identified on the mediastore.	Integer	1	no	
artist_id	The ID of the track artist.	Integer	1	no	library_artists
album_id	The ID of the track album.	Integer	1	no	library_albums
genre_id	The ID of the track genre.	Integer	1	no	library_genres
year	The year of the track.	Integer	0	no	
size	The size of the track, in bytes.	Integer	0	no	
category_id	The ID of the track category.	Integer	1	no	library_categories
composer_id	The ID of the track composer.	Integer	1	no	library_composers
discnum	The disc number of the content. This field is useful for box sets.	Integer	0	no	
titlenum	The title/group number of the CDDA/DVDV/DVDA.	Integer	0	no	
tracknum	The track/chapter number of the CDDA/DVDV/DVDA.	Integer	0	no	

continued...

Field	Description	Type	Default	Nulls?	References
rating	The rating (0 = unkown, 1 = worst, 255 = best). Format specific rating is scaled to 1 - 255 range, for example, 1 stars = 60, 2 starts = 125, ..., 5 starts = 255)	Integer	0	no	
date_added	The date the track entry was added to the **library** table.	Integer	0	no	
date_modified	The date the track entry was modified in the **library** table.	Integer	0	no	
bitrate	The track bitrate.	Integer	0	no	
audio_index	The audio index of the track on the DVD.	Integer	0	no	
format	The format of the track, as defined by the MME_FORMAT_* values.	Integer	0	no	
num_channels	The number of audio channels on the track.	Integer	0	no	
language_id	The ID of the track language.	Integer	1	no	library_languages
samplerate	The sampling rate, in hertz, of the audio stream.	Integer	0	no	
conductor_id	The ID of the track conductor.	Integer	1	no	library_conductors
soloist_id	The ID of the track soloist.	Integer	1	no	library_soloists
ensemble_id	The ID of the track ensemble.	Integer	1	no	library_ensembles
opus_id	The ID of the track opus.	Integer	1	no	library_opus
protected	Indicates if there is DRM on the track	Integer	0	no	
last_played	The date this track was last played by the MME.	Integer	0	no	
fullplay_count	The number of times this track has been played by the MME.	Integer	0	no	
duration	The track length, in milliseconds.	Integer	0	no	
copied_fid	The file ID of the copied file. This field is 0 if the file has not been copied.	Integer	0	no	
playable	Indicates if the track is playable.	Integer	1	no	
permanent	If this field is set to 1, the file cannot be pruned.	Integer	0	no	
inode	Optional. The inode for the associated file.	Integer	0	no	
description	An arbitrary text description of the track.	Text	”	no	

continued...

Field	Description	Type	Default	Nulls?	References
title	The track title.	Text	NULL	yes	
filename	The file name of the media track.	Text	”	no	

Table: library_genres

Primary key: genre_id

Fields:

Field	Description	Type	Default	Nulls?	References
genre_id		Integer		yes	
genre		Text, unique		yes	

Table: library_artists

Primary key: artist_id

Fields:

Field	Description	Type	Default	Nulls?	References
artist_id		Integer		yes	
artist		Text, unique		yes	

Table: library_albums

Primary key: album_id

Fields:

Field	Description	Type	Default	Nulls?	References
album_id		Integer		yes	
album		Text, unique		yes	

Table: library_composers

Primary key: composer_id

Fields:

Field	Description	Type	Default	Nulls?	References
composer_id		Integer		yes	
composer		Text, unique		yes	

Table: library_conductors

Primary key: conductor_id

Fields:

Field	Description	Type	Default	Nulls?	References
conductor_id		Integer		yes	
conductor		Text, unique		yes	

Table: library_soloists

Primary key: soloist_id

Fields:

Field	Description	Type	Default	Nulls?	References
soloist_id		Integer		yes	
soloist		Text, unique		yes	

Table: library_ensembles

Primary key: ensemble_id

Fields:

Field	Description	Type	Default	Nulls?	References
ensemble_id		Integer		yes	
ensemble		Text, unique		yes	

Table: library_opus

Primary key: opus_id

Fields:

Field	Description	Type	Default	Nulls?	References
opus_id		Integer		yes	
opus		Text, unique		yes	

Table: library_categories

Primary key: category_id

Fields:

Field	Description	Type	Default	Nulls?	References
category_id		Integer		yes	
category		Text, unique		yes	

Table: library_languages

Primary key: language_id

Fields:

Field	Description	Type	Default	Nulls?	References
language_id		Integer		yes	
language		Text, unique		yes	

Table: db_sync

The **db_sync** table is used by the generic handler for external database synchronization plugins. It should be considered private to the MME.

No primary key.

Fields:

Field	Description	Type	Default	Nulls?	References
msid	The ID media store that the library table entry is on.	Integer		no	mediastores
fid	The ID of a library table entry that synchronizers have been told about. (*fid*).	Integer		no	library
last_sync	The internal timestamp value when the external database synchronizers were last told about this file.	Integer	0	no	

Table: playlistdata

The **playlistdata** table is available for storing any linear created playlists. They can be selected using the “statement” from the **playlists** table.

Primary key: oid

Fields:

Field	Description	Type	Default	Nulls?	References
oid	An order identifier. This can be used to assign an arbitrary order to the playlist using the SQL ORDER BY clause.	Integer		yes	
plid	The ID of the playlist to which this track belongs.	Integer		no	playlists
fid	The track file ID (*fid*).	Integer		no	library
msid		Integer		no	mediastores

Indices:

Index name	Fields
library_index_folderid_msid_filename	folderid,msid,filename
folders_index_parentid	parentid

Tables in `mme_temp`

Table: nowplaying

The **`nowplaying`** table holds information about the currently playing or last played track for a control context. This information is maintained by the MME: your client application can query it this table, but shouldn't write to it. You can query this table when your client receives a MME_EVENT_TRACKCHANGE event indicating that a new track is playing. The information may be limited by the metadata available, so some fields may not contain data for every track.

The MME doesn't clear this table after a track stops playing, so if there's no playing track, it contains information about the last played track.

Primary key: ccid

Fields:

Field	Description	Type	Default	Nulls?	References
ccid	The ID for the control context where the track is currently playing.	Integer		yes	controlcontexts
playing	Reserved for future use.	Integer	0	no	
fid	The file ID (*fid*) for the track (0 if unknown).	Integer	0	no	library
msid	The ID of the mediastore with the track.	Integer	0	no	mediastores
ftype	The track's file type. See the **ftype** field in the **library** table.	Integer	0	no	
year	The track's year.	Integer	0	no	
bitrate	The track's bitrate, in bytes per second.	Integer	0	no	
samplerate	The track's samplerate, in hertz.	Integer	0	no	
num_channels	The track's number of channels: 1=mono, 2=stereo.	Integer	0	no	
size	The track's size, in bytes.	Integer	0	no	
discnum	The track's disc number.	Integer	0	no	
tracknum	If the track is part of a collection (i.e. an album), the track's	Integer	0	no	
rating	The rating (0 = unkown, 1 = worst, 255 = best). number in the collection.	Integer	0	no	
copied_fid	The file ID for the copied file, placed in the **library** table by media copy and ripping operations.	Integer	0	no	
filename	The filename of the track (empty string if unknown).	Text	”	no	
artist	The track's artist.	Text	”	yes	
title	The track's title.	Text	”	yes	
album	The track's album.	Text	”	yes	
genre	The track's genre.	Text	”	yes	
composer	The track's composer.	Text	”	yes	
conductor	The track's conductor.	Text	”	yes	
soloist	The track's soloist.	Text	”	yes	
ensemble	The track's ensemble.	Text	”	yes	
opus	The track's opus.	Text	”	yes	

continued...

Field	Description	Type	Default	Nulls?	References
category	The track's category.	Text	”	yes	
description	The track's description.	Text	”	yes	

Tables in mme_custom

Table: mediastores_custom

The **mediastores_extra** table is an optional extension to the **mediastores** table. It should have an *msid* column so that it can be joined with the *library* table where *msid=msid*. You should create triggers so that when a row is added to or removed from the **mediastores** table it is also added to or removed from this table.

Primary key: msid

Fields:

Field	Description	Type	Default	Nulls?	References
msid	The mediastore ID.	Integer		yes	

Table: library_custom

The **library_custom** table is an optional extension to the **library** table. It should have an *fid* column so that it can be joined with the **library** table where *fid=fid*, adding this table's columns to the main **library** table. Some examples of columns that could be added to the **library_custom** table are *rating* and *skip_count*. However, any user-defined columns can be added to this table. You should create triggers so that when a row is added to or removed from the **library** table it is also added to or removed from this table.

Primary key: fid

Fields:

Field	Description	Type	Default	Nulls?	References
fid	The file ID.	Integer		yes	

Table: playlistdata_custom

The **playlistdata_custom** table is an sample table placed here to support application created playlists. It has the same fields as the **playlistdata** table. See the description of the **playlistdata** table for information about this table and how it can be used.

Primary key: oid

Fields:

Field	Description	Type	Default	Nulls?	References
plid	The playlist ID.	Integer		no	playlists
fid	The file ID.	Integer		no	library
msid	The mediastore ID.	Integer		no	mediastores
oid	The order identifier.	Integer		yes	

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z