# QNX® Aviage Multimedia Suite 1.2.0

## *MME Developer's Guide*

*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 Developer's Guide* accompanies the QNX Aviage multimedia suite. It is intended for application developers who use the suite's MultiMedia Engine (MME) to develop multimedia applications.

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.

The table below may help you find what you need in this book:

| For information about: | See: |
|---|---|
| Understanding and using control contexts, zones and output devices | Control Contexts, Zones and Output Devices |
| Connecting to the MME and registering for events | Starting Up and Connecting to the MME |
| Working with the MME database | Working with the MME Database and SQL |
| Working with mediastores | Working with Mediastores |
| Synchronizing media | Synchronizing Media |
| Playing audio media files | Playing Media |
| Working with playlists | Playlists |
| Exploring and playing unsynchronized media | Unsynchronized Media |
| Metadata and artwork | Metadata and Artwork |
| Playing and managing video and DVD mediastores | Playing and Managing Video and DVDs |
| Playback errors and how to manage them | Playback Errors |
| Copying and ripping media | Copying and Ripping Media |
| Working with internet streamed media, and with CD changers | External Devices, CD Changers and Streamed Media |
| Working with iPods devices | Working with iPods |
| Working with PFS devices | Working with PFS Devices |
| Working with Bluetooth devices | Working with Bluetooth devices |
| MME sample applications and sample source code | MME Sample Applications |

*continued...*

| For information about: | See: |
|---|---|
| Terminology used in this guide | Glossary |

For an overview of the MME architecture and instructions on how to get the MME started and playing media, see *Introduction to the MME*. For information about the QDB database engine used by the MME and client applications, see the *QDB Developer's Guide*.

Other MME documentation available to application developers includes:

| Book | Description |
|---|---|
| *Introduction to the MME* | MME Architecture, Quickstart Guide, and FAQs. |
| *MME API Library Reference* | MME API functions, data structures, enumerated types, and events. |
| *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** |

*continued...*

| Reference | Example |
|---|---|
| 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

# Control Contexts, Zones and Output Devices

## In this chapter. . .

This chapter describes how to use control contexts, zones and output devices in the MME. For more detailed information about how to start the MME and how to connect to it, see the chapter Starting Up and Connecting to the MME in this guide, and the "MME Quickstart Guide" in the *Introduction to the MME*.

# About control contexts, zones and output devices

To use the MME, you need to define at least one control context in the **`controlcontexts`** table in the MME database. With one control context defined, you can connect to the MME. To play media, you also need to define at minimum one output zone and one output device.

- Control contexts
- Zones
- Output devices

## Control contexts

A client application works with the MME in a *control context*. The MME is a resource manager, and control contexts are the mount points to the MME resource manager. They are responsible for managing requests from the client applications, and for directing other components in the MME and **`io-media`** to complete these requests.

The client application connects to an MME control context in order to be able to create, set and play track sessions, synchronize mediastores, copy and rip files, play tracks, and perform other operations with the MME.

Control contexts are defined statically in the MME database table before the MME starts up. They exist regardless of whether or not clients are connected, and regardless of how many clients are connected. Each control context in the MME has its own thread, so the MME is capable of scaling with as many control contexts as required.

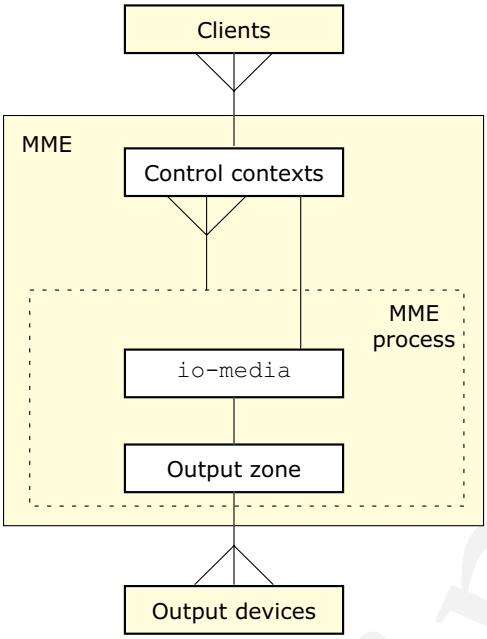
### Client application connections to a control context

Multiple client application connections can be made to a single control context, but a control context will manage only one track session at a time, and will control only one **`io-media`** instance at a time. The **`io-media`** that is controlled by the control context will output to only one output zone at a time.

The figure below shows:

- the one-to-many relationships between an MME process and control contexts, a control context and client application connections, and between an output zone and output devices.
- the one-to-one relationship between a control context and an instance of **`io-media`**, and an instance of **`io-media`** and an output zone.

Note that there is also a one-to-one relationship between control contexts and output zones.




*Client, control context, MME process,* **`io-media`***, output zone and output device relationships.*

The MME's design allows client applications to be built to use only one connection to the MME, or to use multiple connections, with, for example, one connection, "frontseat", to perform all operations, and another connection,"backseats", functioning as a passive output connection that outputs media controlled and played by "frontseat".

The figure on the next page illustrates an implementation of MME with two control contexts in an automobile.

Other possible implementations might be in a home entertainment system where multiple clients connect to a single control context from different interfaces in the house, or an implementation for an aircraft entertainment system that would run the MME in a central location and a control context with an instance of **`io-media`** at every seat to offer passengers play-on-demand music and video.

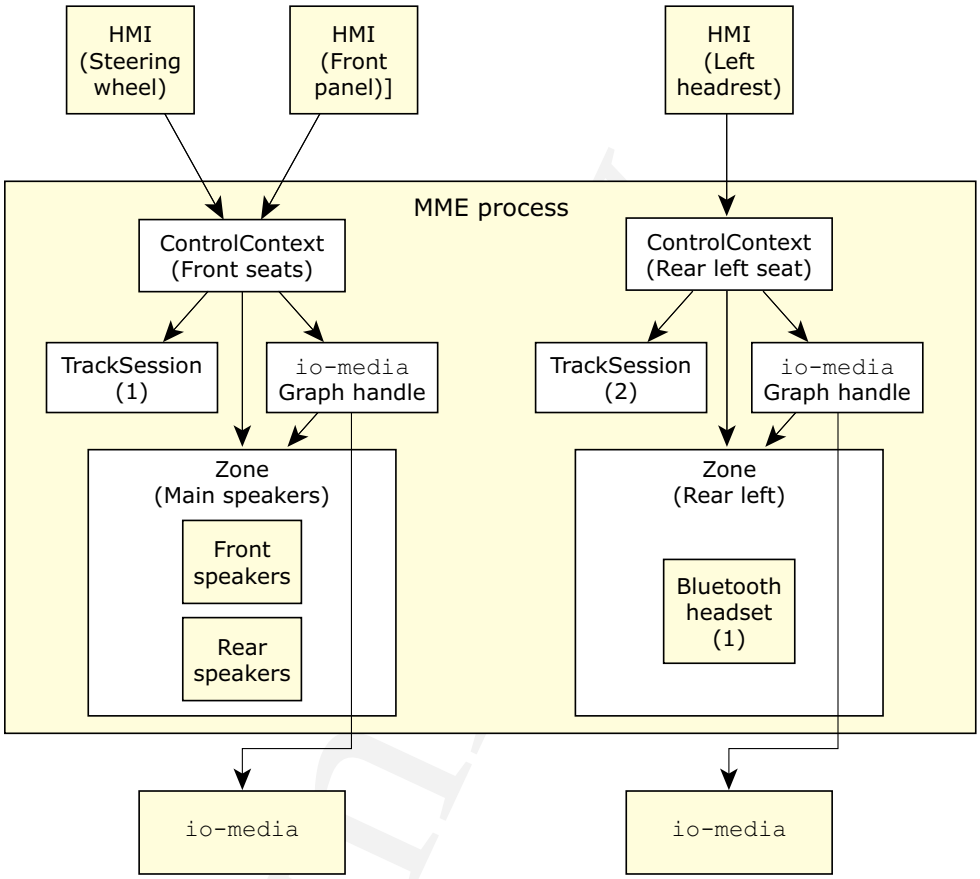



*Illustration of MME implementation with two control contexts.*

### Setting the maximum number of control contexts

The maximum number of control contexts is configured in the MME configuration file **`mme.conf`**. For more information, see "Global settings" in the *MME Configuration Guide*.

## Zones

*Zones* are passive output containers through which the MME sends played media to output devices. Zones can be:

- created at start up
- created or removed as required while the MME is running
- attached to a control context or detached from a control context while the MME is running

For more information, see "Example configurations" and "Runtime control of zones and output devices" below.

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”.

### Output devices

An output device is a device to which media content can be output. Three classes of output can be sent to devices:

- audio content
- video content
- encoded content, which is sent to a remote **`io-media`** for decoding

## Configuring the MME for multi-zone support

The MME uses combinations of control contexts, zones and output devices to play media and direct output to the output locations requested by end-users:

- Control contexts are permanent; they cannot be created or removed while the MME is running. Your MME start up routines should therefore create all the control contexts your implementation will need.
- Zones can be created at start up, but can also be added or removed by calls to the MME API, as required while the MME is running.
  Since you need a zone to which you can send media output, you need to create at least one zone at start up. The only exception to this rule is if you will *only* be routing analog output from an iPod directly to an output device, without passing it through the MME.
- Output devices can be added and removed through the MME API, so you do not need to add them at startup. It is good, practice, however, to attach one output device per zone so that the system is ready for playback and output when it has completed its startup routines.

Since there is a one-to-one relationship between control contexts and zones, a common approach at start up is to create all the control contexts required, a zone for each control context, and attach at least one output device to each zone.

### Sample configurations

The examples below show how to set up control contexts, zones and output devices at startup.

### Example: Defining one control context, zone and output device

The example below shows how to define, in the **mme** database schema, a single control context and one zone with a single output device.

```
INSERT INTO zones( zoneid, name ) VALUES ( 1,'Zone1' );

INSERT INTO outputdevices( type, permanent, name, devicepath )
           VALUES( 1, 1, 'defaultoutput', '/dev/snd/pcmC0D1p' );
INSERT INTO zoneoutputs( zoneid, outputdeviceid )
           SELECT 1, outputdeviceid FROM outputdevices
           WHERE name='defaultoutput';
INSERT INTO renderers( path )
           VALUES( '/dev/io-media' );
INSERT INTO controlcontexts( zoneid, rendid, name )
           VALUES( 1, 1, 'default' );
```

### Example: Defining multiple control contexts, zones and output devices

The example below shows how to define in the **mme** database schema, two control contexts and two output zones with one output device per zone.

```
INSERT INTO zones( zoneid, name ) VALUES ( 1,'Zone1');
INSERT INTO outputdevices( type, permanent, name, devicepath )
           VALUES( 1, 1, 'output1', '/dev/snd/pcmC0D1p' );
INSERT INTO zoneoutputs( zoneid, outputdeviceid )
           SELECT 1, outputdeviceid FROM outputdevices
           WHERE name='output1';
INSERT INTO renderers( path ) VALUES( '/dev/io-media' );
INSERT INTO controlcontexts( zoneid, rendid, name )
           VALUES( 1, 1, 'cc1' );
INSERT INTO zones(zoneid, name) VALUES ( 2,'Zone2');

INSERT INTO outputdevices( type, permanent, name, devicepath )
           VALUES( 1, 1, 'output2', '/dev/snd/pcmC0D2p');
INSERT INTO zoneoutputs( zoneid, outputdeviceid )
           SELECT 2, outputdeviceid FROM outputdevices
           WHERE name='output2';
INSERT INTO controlcontexts( zoneid, rendid, name )
           VALUES( 2, 1, 'cc2' );
```

After you have defined your output zones, you must create them in your control context and attach your output to them. For instructions, see "Runtime control of zones and output devices" below.

For more information about how to configure the MME, see the *MME Configuration Guide*.

# Configuring the MME for multi-node support

The MME can get media and send output to a remote node, as required.

## Getting media on a remote node

The MME uses the MCD (Media Content Detector) utility to detect media content. This utility supports media content detection across a network. To access media on remote devices, you must configure:

- the MCD to detect media at the remote location of your device
- configure the MME's **slots** table with these remote devices and their paths

For information on how to configure the MCD for multi-node support, see "Configuring multi-node support" in the chapter Configuring Device Support of the *MME Configuration Guide*. For more detailed information about the MCD, see the *MME Utilities Reference*.

For more information about configuring the slots table for supported devices, see "Configuring the **slots** table for supported devices" in the chapter Configuring Device Support of the *MME Configuration Guide*.

## Outputting to a remote node

The MME supports output to devices across a network. To output to a device on a remote network node, you need to:

- set the path to the device on the remote node in the **outputdevices** table
- configure the **slots** table for supported devices
- attach that device and its zone to the control context with the playback

You can use an output device that is accessible over Qnet by specifying the full path to the device in your client application, or by setting the device path in the **mme_data.sql**. For example, by changing **/dev/snd/pcmC0D1p** (for a local output device) to **/net...***full path* **.../dev/snd/pcmC0D1p** for a remote output device:

```
INSERT INTO outputdevices(type, permanent, name, devicepath)
    VALUES(1, 1,'defaultoutput', '/net/edosk7780/dev/snd/pcmC0D1p');
```

# Configuring the MME for video support

To configure the MME for video support, you must add the URL of a video output device to the MME's **outputdevices** table, as follows:

```
INSERT INTO outputdevices(type, permanent, name, devicepath)
      VALUES(1, 1, 'defaultoutput2',
      'gf:deviceentry>?param1&param2');
```

- *deviceentry* and its parameters corresponds to the device entry in the **`/dev/io-display`** directory.
- You should also configure **`io-media`** for optimal video performance. For more information, see "Configuring **`io-media`** for optimal video performance" in the *MME Configuration Guide*.
- For information about playing video, see the chapter Playing and Managing Video and DVDs in this guide.

## Adding modifiers to a video output device URL

The URL to the video output device may contain optional modifiers after the question mark ("?"), each modifier separated by an ampersand ("&"), to set default values and behaviors on the device. Supported modifiers are listed below:

| | |
|---|---|
| **`layer`** | Specify the GF layer. |
| **`index`** | Specify the GF display index. |
| **`nsurfs`** | Specify the number of GF surfaces. |

To specify the destination rectangle use:

| | |
|---|---|
| **`dstx`** | Set the destination *x* coordinate. |
| **`dsty`** | Set the destination *y* coordinate. |
| **`dstw`** | Set the destination width, in pixels. |
| **`dsth`** | Set the destination height, in pixels. |

To specify the aspect ratio of the pixels on the physical display use:

| | |
|---|---|
| **`aspn`** | The aspect ratio numerator. |
| **`aspd`** | The aspect ratio denominator. |

To set the color control on the output use:

| | |
|---|---|
| **`sat`** | Specify the color saturation, in direct GF units. |
| **`bright`** | Specify the brightness, in direct GF units. |
| **`contrast`** | Specify the contrast, in direct GF units. |

After playback has started, you can use the *mme_video_set_properties()* function to adjust playback parameters.

### Example: Defining a video output device

The example below shows how to define in the **mme** database schema, an audio output device and a video output device.

```
INSERT INTO outputdevices(type, permanent, name, devicepath)
       VALUES(1, 1, 'rearoutputaudio', 'snd:/dev/snd/pcmC0D3p');
INSERT INTO outputdevices(type, permanent, name, devicepath)
       VALUES(2, 1, 'rearoutputvideo',
       'gf:8086,2772,0?aspn=72&aspd=77&bright=-20&sat=-10');
```

# Runtime control of zones and output devices

This section describes how to manage zones and output devices at runtime.

## Adding and removing zones and output devices

You must set a zone for the control context where you will play media in order to output the playback to an output device. If a zone is no longer required, you can remove it.

### Adding zones to a control context

To create and use a zone, use the following functions:

- To create a zone, call *mme_zone_create()* with the MME connection handle, and the name you want to give the zone.
- To set the output zone for the control context, call *mme_play_set_zone()*.
- To find out which output zone is set for your control context, call *mme_play_get_zone()*.

### Removing a zone

To remove a zone that is no longer required, simply call *mme_zone_delete()* specifying the ID of the zone you want to remove.

### Attaching and detaching output devices

You should also attach output devices to zones, so that the control context will use these devices for playback. To attach a new output device to a zone, use *mme_play_attach_output()*; to detach an output device from a zone, use *mme_play_detach_output()*.

## Making output devices “permanent”

You may wish to mark some output devices, such as built-in car speakers, as permanent, and others, such as removable headphones, as not permanent. To mark an output device as permanently attached to an output zone, call *mme_output_set_permanent()* with the *permanent* argument set to 1 (one). To tell the MME that a device is not permanent, call the same function, with the *permanent* argument set to 0 (zero).

## Managing output attributes

To get output attributes, such as volume, balance, mute, or GF/video layer, call *mme_play_get_output_attr()*. To set these attributes, use *mme_play_set_output_attr()*.

*Chapter 2*

# Starting Up and Connecting to the MME

## *In this chapter. . .*

The information and instructions in this chapter assume that you have installed the MME, and that you have a target system with the MME configured and running. On this target system you need, as a minimum:

- **qdb**, with the base MME databases created (**mme_temp**, **mme**, and **usb**).
- **io-media-generic**
- **io-fs-media** and its modules
- **io-audio** and the correct drivers
- **mcd**
- **mme**

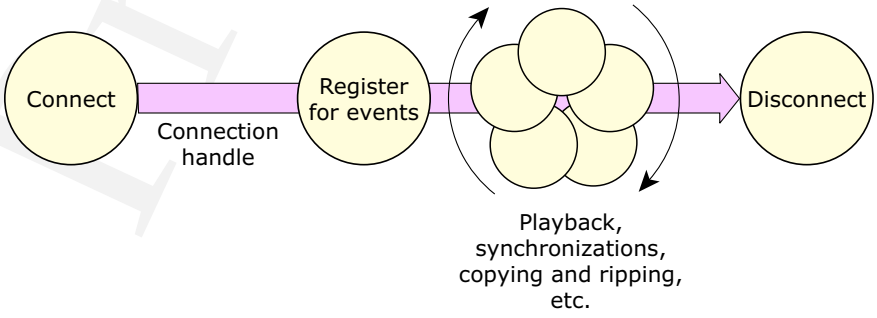
For more information about starting the MME, see the chapter MME Quickstart Guide in *Introduction to the MME*.

In everyday discussions of electronic media, the terms “file” and “track” are often used interchangeably. In this document, “file” refers to all non-media files (the MME configuration file, for instance) and to media files that are being read or otherwise manipulated for a purpose other than playing them. The term “track” refers to media files that are being played or read and otherwise accessed or manipulated for playing. For example, the MME synchronizes folders and the *files* inside them, but it reads the *tracks* from a playlist and places them in a track session.

# Connecting to the MME

The MME is designed to make communications with client applications both simple to implement and efficient in its execution. To communicate and work with the MME, a client application needs only to connect to the MME and register for the types of events it requires for that connection.

The figure below illustrates the flow of activities from first connection to the MME to disconnection.




*Client application workflow with MME from connection to disconnection*

You may also want to set the preferred language for media output. For more information, see "Setting the preferred playback language" in the chapter Configuring Internationalization of the *MME Configuration Guide*.

- For information about control contexts and how to define them, see "Control Contexts" in the chapter Control Contexts, Zones and Output Devices.
- For information about detecting mediastore states at startup, see "Understanding mediastore states at startup" in the chapter Working with Mediastores.

## The MME connection handle

Each client application connected to the MME has its own unique connection handle. The connection handle information is stored in the opaque structure **`mme_hdl_t`**. Valid connection handles are created by *mme_connect()*. The MME fills in all needed information to create the connection handle; 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 maintain thread safety.

### About connections, validation and blocking

The MME allows you to set a timeout period for blocking functions. If you set a timeout period, when the timeout period expires, the function returns, unblocking the client application. To enable the MME's unblocking capabilies, you need to set the **`<Unblock>`** configuration element attribute to **`true`**. For more information, see "Enabling the unblock capability" in the the *MME Configuration Guide*, and *mme_set_api_timeout()* and *mme_get_api_timeout_remaining()* in the chapter MME API.

For information about setting the MME's behavior when making a connection (synchronous or asynchronous, and blocking or non-blocking) see *mme_connect()*. For blocking and validation information for specific functions, see the descriptions of the functions in the *MME API Library Reference*.

## Making the connection

Before it can start using the MME API, your client application must:

- use *qdb_connect()* to connect to the MME database
- use *mme_connect()* to connect to at least one MME control context

For more information about QDB, see the *QDB Developer's Guide*.

The function *mme_connect()* requires:

- a control context device path for your connection (which maps directly to a device path)
- settings for the *flags* variable

For more information, see *mme_connect()* in the *MME API Reference*.

## Disconnecting from the MME

When the client application has finished all the work it needs to do on an MME connection, it should disconnect from the MME and the QDB. To disconnect from the MME and the QDB, call the functions *mme_disconnect()* and *qdb_disconnect()* with the appropriate connection handles.

The sample below shows how a client application disconnects from the MME and the QDB:

```
// disconnect from the servers.
mme_disconnect( mme );
qdb_disconnect( qdb );
```

Disconnecting from the MME doesn't shut down the MME. It simply disconnects the client application from the device path to which it was connected in a control context.

# Shutting down the MME

To shut down the MME:

1. Call *mme_shutdown()* to prepare the MME for shutdown.
2. Call *mme_disconnect()* to disconnect from the MME.
3. Kill the **mme** process, or shut down the system.

A call to *mme_shutdown()*:

- stops and disables:
  - playback on all control contexts
  - synchronizations on all control contexts
  - any other MME operations that write to the MME database

- delivers to all control contexts, the events MME_EVENT_SHUTDOWN when the MME begins shutting down threads in the background, and MME_EVENT_SHUTDOWN_COMPLETED when all threads have shut down

Because the MME shuts down threads in the background, the client application may receive events from other operations after it receives MME_EVENT_SHUTDOWN and before it receives MME_EVENT_SHUTDOWN_COMPLETED.

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.

A call to *mme_shutdown()* disables the MME. The MME must be killed and restarted before a client application can use it again.

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.

Before calling *mme_shutdown()*, make sure that your client application completes necessary operations and, if necessary, informs the users that it is shutting down.

# Using the MME notification interface

The MME uses events to communicate with client applications. Client applications should be designed to use these events to trigger processes, from responding to end-user input to handling errors.

In order to receive events from the MME, the client application must:

- register for events, specifying the classes of events it wants to receive
- request the events at the appropriate times

## Registering for events

To receive events after connecting to the MME, a client application must use *mme_register_for_events()* to register for events, 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 connections for different classes of events. For example, a connection used to handle synchronizations can register for synchronization events, but ignore 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 **`sigevent`**s 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.

For information about , see **sigevent** in the *Neutrino Library Reference*.

The client application can then decide from the **sigevent** if it needs to see the associated event.

The code snippet below provides an example of how to register for events in a Photon environment. It shows the steps required to set up the client application and the MME so that the MME delivers playback events (MME_EVENT_CLASS_PLAY) to the client application. These steps are:

- creating a pulse
- attaching the client application's input handler function *my_input_handler()* to the pulse
- filling the structure **mme_event** with the relevant event data
- instructing the MME to place the event data for each playback event in its event queue

The client application now needs only to call *mme_get_event()* to retrieve playback events with their data.

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

...

// Set up the MME input handler.
if( 0 == ( pulse = PtAppCreatePulse(NULL, -1)) ) {
    perror( "PtAppCreatePulse()" );
    exit( EXIT_FAILURE );
}

if( NULL == PtAppAddInput( NULL, pulse, my_input_handler, NULL) ) {
    perror( "PtAppAddInput()" );
    exit( EXIT_FAILURE );
}

PtPulseArm(NULL, pulse, &mme_event);

// Let the mme know that we need events for this class.
if( -1 == mme_register_for_events(
    mme, MME_EVENT_CLASS_PLAY, &mme_event ) ) {
    perror( "mme_register_for_events()" );
    exit( EXIT_FAILURE );
}
```

The code snippet below shows how the MME's command-line tool **mmecli** registers for events.

```
// Connect to the mme for the event thread
mme_ev_hdl = mme_connect(controlcontextdevice, O_SYNC);
if (mme_ev_hdl == NULL) {
```

```
    fprintf(stderr,
        "Could not connect to %s\n", controlcontextdevice);
    exit(EXIT_FAILURE);
}

// We need a channel to receive the pulse notification on.
chid = ChannelCreate(0);

// And we need a connection to that channel for the pulse
// to be delivered on.
coid = ConnectAttach(0, 0, chid, _NTO_SIDE_CHANNEL, 0);

// fill in the event structure for a pulse
SIGEV_PULSE_INIT(&event, coid,
    SIGEV_PULSE_PRIO_INHERIT, MY_PULSE_CODE, 0);

// Setup the timer; we want first event right away.
timer_create(CLOCK_REALTIME, &event, &timer_id);
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = 0;
itime.it_interval.tv_sec = 0;
itime.it_interval.tv_nsec = 0;
itime.it_interval.tv_sec = 0;

// Register for all events from the MME
if (mme_register_for_events(mme_ev_hdl,
        registeredclasses, &event) == -1) {
           fprintf(stderr,
           "Could not register for events of type ALL. errno = %d\n", errno);
    finish(0);
}
```

## MME event classes

The MME event classes are bit masks. You can combine them with an OR operator to register for several events at once. The structure **`mme_event_classes_t`** defines the different MME event classes as bit masks. These classes are:

- 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.

To register for *playback* and *synchronization* events call *mme_register_for_events()* as follows:

```
mme_register_for_events( hdl,
                         MME_EVENT_CLASS_PLAY | MME_EVENT_CLASS_SYNC,
                         event);
```

## Getting events

To see the events in the MME's event queue, your client application needs to call *mme_get_event()*. The example below shows part of a routine used by a client application to get events from the MME:

```
for(;;) {

    if( -1 == mme_get_event( mme, &event ) ) {
        perror( "mme_get_event()" );
        return Pt_CONTINUE;
    }

    if( event.type == MME_EVENT_NONE ) {
        break;        // no more events, exit the loop.
    }
}
```

For a complete list of events delivered by the MME, see the chapters on events in the *MME API Library Reference*.

## Unregistering 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 *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. The example below shows a registration to receive all event classes, and a registration to stop receiving media copy and ripping events:

```
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 and QDB `slog` codes

The MME and QDB slog codes have permanent values, as follows:

- _SLOGC_QDB — 26
- _SLOGC_MME — 27

*Chapter 3*

# Working with the MME Database and SQL

## *In this chapter. . .*

# Time values in the MME database

The MME's time is a 64-bit, internally-derived value that is guaranteed to be monotonically increasing, even across system restarts. This value is guaranteed on systems with or without a Real-Time Clock, and on systems on which the real-time is changed forward or backward.

This behavior permits time-based comparisons of entries in the database with other database entries, such as, for example, the *lastseen* and *last_sync* fields in the **`mediastores`** table to determine if a mediastore requires resynchronization.

The table below lists MME database table columns (fields) that use the MME's internally derived time. These fields can be compared to determine the relative sequence of events, as in the example above.

| Table | Columns (Fields) |
|---|---|
| **`folders`** | |
| **`library`** | |
| **`mediastores`** | *lastseen* and *last_sync* |

# Solutions for database deadlock issues

Database deadlock issues have been observed on some MME projects. The causes for these issues have been identified, and the solutions are described below:

- Different database file attached orders
- Using the QDB client to verify attached order
- Separating deadlock issues from performance issues

## Different database file attached orders

Different database file attached orders for QDB and an external SQLite client result in different locking orders, which cause database deadlocks.

### Solution

To prevent database deadlocks caused by different database file attached orders, ensure that your projects lock databases in the same order as they are attached:

1. **`mme`** (master)
2. **`mme_temp`**
3. **`mme_custom`**
4. **`mme_library`**

If you don't have an **mme_custom** table, use this order:

1. **mme** (master)
2. **mme_temp**
3. **mme_library**

**CAUTION:** Locking your database files in any other order causes database deadlocks.

## Using a QDB client to verify attached order

Before attaching database files in an external external client, you can have the client ask QDB the attached order for the files. Below is an example of how to ask QDB the attached order of database files, and the result:

```
qdbc -d mme 'pragma database_list;'
Rows: 5  Cols: 3
Names: +seq+name+file+
00000: |0|main|/fs/tmpfs/mme.db|
00001: |1|temp||
00002: |2|mme_temp|/fs/tmpfs/mme_temp|
00003: |3|mme_custom|/fs/tmpfs/mme_custom.db|
00004: |4|mme_library|/fs/tmpfs/mme_library.db|
```

If a file doesn't have a filename (row 1), then don't attach it.

## Separating deadlock issues from performance issues

During the development phase of your project you should configure your systems to ensure that you are able to correctly separate performance problems from deadlock problems, and understand and solve each problem accordingly:

- Run your systems with infinite timeouts to ensure that a deadlock is not confused with a performance issue, and is always correctly identified and addressed.
- Enable profiling for queries that take longer than a specified time to execute (for example, 200 milliseconds). If a query takes longer than the specified amount of time, log it as a performance warning, and address the performance issue.

You can change your system configuration when you prepare your system for the production environment.

# Handling of corrupt database

If an operation that uses SQLite, such as those performed by *qdb_statement()* or *qdb_vacuum()*, fails because of a corrupt database, the function now returns EBADF

and logs an error. Client applications can now check for EBADF, and take appropriate steps to correct the problem with their databases.

For information about checking for and correcting inconsistencies, see the information provided with the *mme_sync_db_check()* function in the *MME API Library Reference*.

# Optimizing your SQL

This section provides a few tips on how to optimize your SQL when working with the MME.

SQL is very flexible and can perform the same job in many different ways. Not all SQL statements are equal, however, and it is important to optimize your client application's requests to the MME database. SQLite is fast, but it can take time to complete an operation if the query statement is not optimized.

For an overview of how to optimize SQL statements for SQLite, see "SQLite Optimizer Overview" on the SQLite web site **`www.sqlite.org`**.

#### A note about SQL statements

The QDB (**`qdb`**) resource manager is a resource manager interface on top of the SQLite database engine. Through the QDB, the MME uses SQLite to query and write to its databases. This section offers recommendations for composing queries and other SQL statements for the MME.

SQL statements are *not* case sensitive. For example, the three queries below are equivalent:

```
select fid,msid,filename from library

SELECT fid,msid,filename from library

SELECT fid,msid,filename FROM library
```

By convention, however, we use capitals for the SQL keywords to improve the legibility of query statements: **`SELECT fid,msid,filename FROM library`**.

## Design for size and limit queries

An SQL database can become very large very quickly, with hundreds of thousands of entries. The MME database is designed to scale well, but it's best to limit your queries and to design these queries to avoid duplicating information in the database tables.

## Use Indexes

Indexes improve database performance. When a query is made against a table, if a column doesn't have an index, it requires a table scan; and if an unindexed column is of type TEXT, SQL will perform a full table scan string comparing all rows with the requested value.

## Use JOINs carefully

Joins are convenient, but they don't scale well and are often much slower than sub-selections for large tables, because the complexity of JOINs is exponential, while the complexity of sub-selections is linear. As you add more rows to the tables, the query sub-selection will increasingly perform better than the query with the JOIN.

The following examples produce the same results, but the statement with the sub-select is much faster, especially with larger tables.

### Not recommended

```
SELECT fid FROM library
    INNER JOIN mediastores on library.msid = mediastores.msid
    WHERE mediastores.available = 1;
```

### Recommended

```
SELECT fid FROM library
    WHERE msid IN (SELECT msid FROM mediastores WHERE available=1);
```

If you join two small tables that will never be large, then using a JOIN is acceptable, as it won't impact performance. However, the query with the JOIN won't scale well and performance will cause performance to degrade if either one of the tables increases in size.

## Filtering out unavailable tracks

Media files on external devices, such as a PFS device, remain in the MME library after the device has been removed from the system. The exception to this rule is if the device and its files are pruned from the library to keep the MME database within its configured size limits. For information about database pruning, see "Database pruning" in the *MME Configuration Guide*. In addition, files that were synchronized but are later found to be unplayable remain in the library, though they are marked as unplayable. For more information, see "Marking unplayable tracks" in the chapter Playback Errors.

To avoid building track sessions with tracks that aren't available, which could cause gaps in playback, your client application should filter out tracks on unavailable mediastores when it builds its track sessions. It should include either **`WHERE available=1`** or **`WHERE active=1`** in its select statement. The example query statement below selects all tracks in the playlist "Favorites" that are on available mediastores:

```
SELECT fid FROM playlistdata WHERE
    plid = (SELECT plid from playlists WHERE name = 'Favorites')
    AND msid IN (SELECT msid FROM mediastores WHERE available=1);
```

# Chapter 4

# Working with Mediastores

## *In this chapter. . .*

This chapter describes how to detect and manage mediastores, their state changes, and their removal from and insertion into the MME.

# Detecting mediastores

The MME uses the Media Content Detector (MCD) utility to monitor and detect the insertion and removal of mediastores, and the **`slots`** table to associate mediastores with the devices that present them. To detect mediastores, your system must have the MCD *and* the slots table configured correctly. For more information, see "Mediastore detection path configuration" in the the chapter Configuring Device Support of the *MME Configuration Guide*..

## Mediastore states

Mediastores can have any one of the following states:

- Nonexistent — the MME has no database entry for the mediastore.
- Unavailable — the MME has a database entry for the mediastore, but the mediastore isn't in the system in which the MME is running.
- 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 can't be synchronized, and tracks on the mediastore can't be ripped or played. This state is generally possible only for disk-based mediastores in multidisk changers.
- 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.

The initial state of all mediastores is "nonexistent". The MME checks for the insertion and removal of mediastores, including hard drives, CDs and DVDs, and USB memory sticks, and delivers the event MME_EVENT_MS_STATECHANGE when a mediastore state changes.

When the state of a mediastore changes from another state to "nonexistent", the MME prunes the entries for that mediastore from its database, *regardless* of the MME's pruning settings.

The default MME configuration is to automatically detect new mediastores. When it detects a new mediastore, the MME:

- checks if it has seen the mediastore before, by attempting to match a unique mediastore identifier with an entry in the *identifier* column of the MME **`mediastores`** table
- updates the **`mediastores`** table and sets
  - the *available* field for the mediastore to indicate that the mediastore is available

- if the mediastore is active, the *active* field to indicate that the mediastore is active

- delivers the event MME_EVENT_MS_STATECHANGE

### Understanding mediastore states at startup

In order to start the MME correctly, you should keep in mind the following:

- The MME database must be accessible to the MME before it starts. This requirement means that the QDB must be running before the MME is started. Note, however, that with the QDB running, the MME databases can be read by other entities, including client applications that use the MME.
- The MME can't use a mediastore that it hasn't been told about by the path monitoring system.
- When the MME first starts, it has no way of knowing:
  - what mediastores are present
  - what changes were made to mediastores or to its database while it was not running — or the significance of those changes
- The mediastore information in the MME database may vary, depending on how your system is configured, and how it shutdown; and you should handle the system startup accordingly:
  - If your system is configured to always start from a clean (empty) database at each startup, you only need to tell the MME to begin device detection.
  - If your system did not perform a clean shut down (for example, a power failure or a battery removal stopped the system), you need to revert to a clean database, and proceed from that point.
  - If your system is configured to save the database at shutdown and restore it at startup (the recommended configuration), the state of mediastores indicated by the MME database at startup is *the state of the mediastores when the MME shut down*. If mediastores were removed, inserted or otherwise changed between shutdown and startup, these changes are not indicated in the database, so you need to tell the MME to begin device detection, then *wait for it to complete its database clean up before attempting to access the database*. See "System startup operations" below for details.

For more information about mediastore states, see "Detecting mediastores" in the chapter Working with Mediastores.

#### System startup operations

The following describes the operation of the system at startup, *assuming* that automatic device detection is disabled.

**1** The QDB is started.

- The database may be read.

- The **mediastores** table reflects the state of mediastores at the previous shutdown, assuming that the table has been properly restored and no other entity is writing to it.

2. The MME is started. Since device detection is disabled, the **mediastores** table doesn't change state, and the client application has a second opportunity to determine the startup (in fact, the previous shutdown) state of the mediastores.
3. The client application instructs the MME to begin device detection. The MME goes through the **mediastores** table, and sets any **active** or **available** entries to **unavailable**. For each changed entry, the MME delivers an MME_EVENT_MS_STATECHANGE event. During this time, the **mediastores** table should not be read by other entities, as it may be changing.
4. The MME delivers an MME_EVENT_MS_DETECTION_ENABLED event. Delivery of this event indicates that the MME has completed its database clean up. The client application may now read the database.

**CAUTION:** The client application should not read from the MME database until the MME completes its database clean up and delivers the MME_EVENT_MS_DETECTION_ENABLED event. Until this event is delivered, the database reflects the *previous* state of mediastores, creating an inconsistency between the information about the mediastores in the MME database and the actual state of the mediastores. This inconsistency can cause errors.

5. The MME handles insertion requests. After the MME delivers an MME_EVENT_MS_DETECTION_ENABLED event, it may process mediastore insertion requests from the path monitoring system. Handling of these causes the database to change: as the path monitoring system (normally the MCD) detects the appearance of media stores, it tells the MME, and the MME processes this new information and updates the **mediastores** table as needed.

If the MME is configured for *automatic* device detection, the MME executes Step 3 internally, and the states for mediastores in the database state may change between Step 2 and Step 4 above.

#### Determining mediastore state changes after shut down

In some situations, a client application may want to determine if a mediastore remained in the system (was *not* removed) while the system was shut down. This information depends on information to which the MME does not have access. For example, to tell the client application that a CD remained in the system, the MME would need to know if CDs can be removed and inserted while the system power is off.

The mediastore information from the MME database that is directly available to the client application is the following:

- The mediastore state at the previous shutdown. The client application may read this information from the MME database *after* the QDB has started, and *before*:

  - the MME is started, if automatic device detection is *enabled* (the default)
  - device detection is started, if automatic device detection is *disabled*

- Mediastore state changes at system startup.

These limitations mean that, in order to be able to distinguish between a mediastore that was never removed from the system and a mediastore that was inserted just as the system was starting up, the client application must be designed to use information that it requests and maintains independently of the information the MME can provide. If, for example, the client application is implemented in a system where CDs cannot be ejected when the system is shut down, it may be able to assume — independently of the information provided by the MME — that a CD that was present at shutdown is present at startup.

#### Configuring how the MME handles mediastores at startup

The *delete_at_start* field in the **`slots`** table allows you to manage how the MME processes mediastores marked as **`active`** at startup.

The default MME behavior at startup is to change the state of **`active`** mediastores to **`unavailable`**. However, when the *delete_at_start* field for a slot is set to a non-zero value, at startup the MME marks any mediastore found in the slot for deletion, and sets its state to **`non-existant`**. This configuration causes the MME to treat all mediastores in a slot as new mediastores; that is, as mediastores that the MME has never seen, and to perform synchronizations accordingly.

## CD detection and presentation

The stages of disk insertion and detection are:

1. Disk inserted into drive.
2. The disk spins up.
3. The MCD notices the path appearance for the CD.
4. The MCD signals the insertion to the MME.
5. The MME probes the disk to see how to handle it.
6. The MME creates a media store entry (or marks an existing entry of the disk as active).
7. The MME attempts to synchronize its database with the contents of the disk.

### Mixed-mode CDs

The MME uses the first entry in a CD's table of contents (TOC) to determine if the CD is an audio or a data CD, and makes only one entry in the **`mediastores`** table for the CD. This behavior means that the MME presents a CD with both audio and data files to the client application as either an audio CD or a data CD, based on the type of file in its first TOC entry.

## Recommended method for detecting mediastores

The recommended method for an client application to detect a mediastore state change, such as an insertion or a removal, is to check for the MME_EVENT_MS_STATECHANGE event, then check the new mediastore state:

- If the new state is e_mme_ms_active, you can assume that the mediastore has been inserted.
- If the new state is e_mme_ms_unavailable, you can assume that the mediastore has been removed.

Note that a state of e_mme_ms_nonexistent can occur when a mediastore has been removed from the MME database, for example, during a pruning operation. Client applications should therefore also check for this state.

## Manually requesting device and mediastore detection

If you have configured the MME not to automatically detect devices and mediastores, you must call *mme_start_device_detection()* to start device and mediastore detection. The MME will check for any new devices that may have or be mediastores, and updates its **`mediastores`** table with the relevant information.

**CAUTION:** Between the time the MME starts up and mediastores are detected, the MME can't check the state of mediastores as defined in its database against the actual state of mediastores connected to the system. Therefore, 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.

## Mediastore identifiers

When the MME detects that a mediastore has been inserted into the system, it checks the mediastore for a unique identifier that it can match against an entry *identifier* column of the MME **`mediastores`** table. If the unique identifier for a mediastore matches an entry in this table, the MME considers that it has seen the mediastore before and proceeds accordingly; it may, for example, be able to optimize the synchronization of the mediastore if it ican confirm that some of the information it has about the mediastore is still accurate.

### Identifiers for hard drive filesystems, USB memory sticks, and data CDs and DVDs

When a mediastore is inserted, if the MME has sufficient information to do so, it identifies that mediastore as already known. When it detects the insertion of a hard drive filesystem, USB memory stick, or data CD or DVD into the system, the MME searches for the file **`WMPInfo.xml`** at the root of the mediastore. It then attempts to extract the *UUID* from the file. If the extraction is successful, the MME stores this

*UUID* in the *identifier* column of the MME **`mediastores`** table and uses it as a unique identifier for the mediastore.

If the MME doesn't find the file **`WMPInfo.xml`**, or if it is unable to extract a *UUID* from the file, it creates an identifier from a hash of the volume name (if found) and some file system information.

Note that in the absence of a *UUID*:

- Data CDs or DVDs that have changed their content since the last time they were inserted in the system are recognized as new mediastores.
- Data CDs or DVDs with the same volume name are recognized as different if the size of their contents is different.
- USB device serial numbers can't be used, so there is a significant chance that these devices can't be uniquely identified: two USB devices of the same size with no volume name aren't distinguishable.

### Identifying USB memory sticks

Many USB memory sticks don't have at least one of a **`WMPInfo.xml`** file, a volume name or a unique serial number. Without any of these unique identifiers, the MME has no mechanism for distinguishing between two USB sticks of the same size.

To solve the problem in a development environment, you can either assign a unique volume name to each USB stick, or synchronize each stick with Windows Media Player, which automatically creates a **`WMPinfo.xml`** file. In a production environment, you can have the HMI write a volume name or other unique identifier to USB sticks the first time they are inserted.

### Support for multiple instances of a mediastore

The MME supports up to 10 instances of the same mediastore. When it detects a mediastore, the MME checks if the an instance of that mediastore is already present in its database. If the new mediastore is a duplicate, the MME:

1. Doesn't change any information for any instances of the mediastore already present in the MME database.
2. Creates an entry for the new (duplicate) mediastore.
3. Appends "-i*n*", where *n* is the instance number, to the string in the mediastore's *identifier* column, and (if this column is not empty) to the string in the *driver_identifier* column.

Thus, for example, if two duplicates of a mediastore with *msid* 123 are entered in the system, the MME **`mediastores`** will have three entries for this mediastore, as follows:

| *msid* | *identifier* | *driver_identifier* |
|---|---|---|
| 123 | FAT12341234 | 6 |
| 123 | FAT12341234-i1 | 6-i1 |
| 123 | FAT12341234-i2 | 6-i2 |

- Identifiers can be from “-i1” to “-i9”.
- The MME treats each mediastore instance as a separate, unique mediastore.
- The MME prunes unused mediastore instances from its database, as required.

## Mediastore and device capabilities

The MME uses the *capabilities* field in the **`mediastores`** table to store the capabilities of a mediastore or a device, such as an iPod, that presents itself as a mediastore. This field is a bit map defined by the MME_MSCAP_* constants.

To find out the capabilities supported by a mediastore or device, after receipt of an MME_EVENT_MS_STATECHANGE event indicating that the mediastore or device is *active*, check the value of the *capabilities* field for that mediastore or device.

If, for example, a device manages its own:

- track sessions, the bit for MME_MSCAP_DEVICE_TRACKSESSIONS (0x00000080) will be set
- repeat and random modes, the bit for MME_MSCAP_DEVICE_REPEATRANDOM (0x00000800) will be set

For more information about mediastore and device states, see “Mediastore states” above.

### Track session capabilities

To get details about the current track session capabilities, call *mme_trksession_get_info()*. Note, however, that the information provided by this function is valid only if it is retrieved *after* playback has started on the external device.

# Mapping mediastore filesystem paths to device locations

In order to be able to associate a filesystem path to the physical location of a mediastore, your client application should map the filesystem paths of mediastores to device paths, and these device paths to the physical locations of devices. The *mountpath* field of the **`mediastores`** table is always the filesystem path of a mediastore. To map the mountpaths in the **`mediastores`** table to device paths and, finally, to the physical locations of devices, your client application must know the following:

- what physical devices are in the system (e.g. CD changer in the front seat and a CD changer in the back seat)
- the device paths of the drivers used to handle these physical devices (e.g. **/dev/cd_front** and **/dev/cd_back**)
- how filesystems of mediastores are mounted when they are found (e.g. **/fs/cd_front** and **/fs/cd_back**)

With this information, your client application could map, for example, **/fs/cd_back** from the *mountpath* field in the **mediastores** table to the device path**/dev/cd_back** and know that this mediastore is in the back seat CD changer.

If Qnet is running and the MME is handed the device path, the filesystem mountpath found in the mediastores table will be prefixed by **/net/***nodename*.

### Associating devices and mediastores in the `slots` table

The **slots** table is used to associate mediastores in the MME system with the devices that provide them. As such, a slot is a representation of a device, such as a CD drive or a USB stick. The MME must have an entry in its **slots** table with the mountpath (or, in some cases, the device path) of every device that it may encounter. If the **slots** table doesn't have an entry for the device, the MME will not recognize the device and will not find the mediastores on that device.

The **slots** table is preloaded with default entires for an HDD, as well as for CD/DVD, USB, PFS, UPnP, and iPod devices. You should review these entries and modify, add or delete entries in the table to match your system. For instructions, see "Configuring the slots table for supported devices" in the chapter *MME Configuration Guide*.

## Handling external disk changers

The MME fills in the *name* field in the **mediastores** table when the state of an external CD changer is set to **available**. This behavior allows the client application to communicate the CD changer name to users as soon as the changer is detected and available, even before it is active.

To trigger this new behavior, the client application must configure the **slots** table *name* fields corresponding to external CD disk changers to either empty strings or NULL values.

For other mediastores, the MME sets the *name* field in the **mediastores** table when the mediastore state is set to **active**.

# Handling removed mediastores

When the MME detects that a mediastore has been removed from the system, it:

- sets the *available* and *active* fields for the mediastore in the **`mediastores`** table to **`0`**
- Delivers the event MME_EVENT_MS_STATECHANGE

Note that the *available* field set to **`0`** indicates only that the mediastore is not available. It does not provide information about the state of the mediastore synchronization.

**CAUTION:** If a mediastore is removed from the system while the MCD that monitors it is not running, the MME will *never* learn that the mediastore has been removed.

# Handling reloaded mediastores

If the mediastore was inserted in the system and synchronized at a previous time, the content of the **`library`** table will have the mediastore information. However, because there can be no guarantee that a mediastore exactly matches the information in the MME database, the MME must resynchronize the mediastore to either confirm that all information is accurate, or add, remove, and change information as required.

During this resynchronization process, the MME uses any information available on the mediastore that can confirm the accuracy of its database before all synchronization work is complete. This strategy allows the MME to optimize performance by skipping resynchronization of parts of the mediastore that are confirmed unchanged since the last insertion and synchronization.

The default MME behavior is to automatically synchronize mediastores. Your client application needs only to monitor the progress of the synchronization to know when it can start playing media, displaying metadata, and using playlists.

# "Manually" updating the `library` table

The MME provides a function, *mme_lib_column_set()*, that allows you to "manually" set the values of some fields for mediastores in the **`library`** table.

The function *mme_lib_column_set()* can only be used to update entries in the columns listed below:

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

- *copied_fid*

For more information, see the *MME API Library Reference*.

# Chapter 5

# Synchronizing Media

## *In this chapter. . .*

This chapter explains how to synchronize media on these mediastores with the MME database.

This chapter describes the MME default behavior. For information about how to configure the MME's synchronization behavior, see the chapter Configuring Media Synchronizations in the *MME Configuration Guide*.

For information specific to synchronizing iPods, see "Synchronizing iPods" in the chapter Working with iPods.

# The synchronization process

Mediastore synchronization is the process by which the MME examines mediastores and updates its database with information about the media tracks on the stores and with the metadata for these media. Information and metadata includes, but is not limited to, media type and format (audio, video, etc.), track name and language, genre, cover art, and so on. This information and metadata is essential for the MME and client application to be able to find, organize and play media, and to display meaningful information to the end user.

The MME can be configured to automatically synchronize mediastores on system startup and on mediastore insertation, or to wait for requests to synchronize a mediastore. The default MME configuration is to automatically synchronize all mediastores except iPods. Automatic synchronization is always disabled for iPods; the MME only synchronizes these devices when it is explicitly requested to do so.

## Synchronizer selection

When it prepares to synchronize a mediastore, the MME selects the most appropriate synchronizer for the mediastore. The selection criteria include ensuring that the MME obtains the most accurate and complete metadata available for the files on the mediastore. For example, for a CDDA:

- The MME checks if the CD device supports CD-Text, and if the Gracenote plug-in is enabled.
- If CD-Text or Gracenote support is available, the MME uses the most appropriate synchronizer to get metadata during the metadata synchronization pass.
- If these synchronizers are not available, the MME uses its default synchronizer to get the metadata.

### Metadata synchronizer selection

The MME includes a table, **`metadataplugins`**, that lists the different metadata synchronizers available to the MME. Its fields are:

*metadatapluginid* The metadata plugin ID.

*name* The name of the metadata plugin.

The **`mediastores`** table implements the field **`metadatapluginid`** to identify the metadata synchronizer used for the mediastore, and thereby identify the origin of the metadata for the mediastore. If more than one metadata synchronizer is required for the mediastore, the **`metadatapluginid`** field in the **`mediastores`** table is set to 0 (zero).

For information about configuring ratings for metadata synchronizers, see "Metadata synchronizer ratings" in the *MME Configuration Guide* chapter Configuring Metadata Support.

## Multiple synchronization passes

For most mediastores, the MME uses a multiple synchronization passes process. This multiple pass process reduces the delay time between the insertion of a mediastore and readiness to play media by separating synchronization into separate passes, as follows:

- file and folder discovery
- metadata update
- playlist compilation
- external database syncrhonization (future implementation)

### Monitoring synchronization progress

The client application can register to receive synchronization events and use these events to monitor the progress of the MME synchronization activities. These events tell the client application what level of information is ready for use:

MME_EVENT_MS_SYNCFIRSTFID

The MME has found the first playable track on the mediastore.

MME_EVENT_MS_UPDATE

An MME synchronization process has updated a database table: synchronization is progressing normally.

MME_EVENT_MS_1PASSCOMPLETE

Basic file information: the media can be played.

MME_EVENT_MS_2PASSCOMPLETE

Metadata: artist name, genre, album art, etc. is ready for display to the end user.

MME_EVENT_MS_3PASSCOMPLETE

Playlists: playlists are ready for display and use.

MME_EVENT_MS_SYNCCOMPLETE

All synchronization passes for the mediastore are complete.

## The synchronization pass process

Each synchronization pass proceeds as follows:

1. Start at the root node of a tree (in this case, the mediastore root directory).
2. Enumerate each object in the root node before examining the contents of any of the objects.
3. Examine the first folder in the queue, following each branch to its end before starting the next folder.

This process ensures that all directories that share a common parent node are synchronized before directories deeper in the tree are examined. If you cancel a synchronization in progress, some directories may be fully synchronized while others may not have any of their contents synchronized.

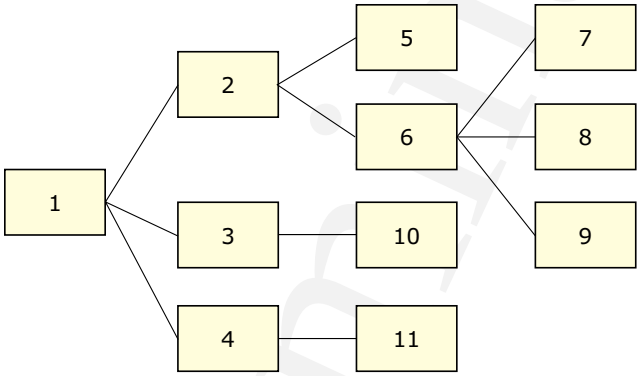



*Illustration of the order in which the MME synchronizes mediastores.*

## Tracking mediastore synchronization status

The MME maintains synchronization flags in its database to track the history of synchronization status of mediastores and folders. The column, *syncflags*, in the **`mediastores`** table is used to indicate which synchronization passes have been completed on a particular mediastore. It consists of three-bit fields, where a set value means the particular synchronization pass has been successfully completed:

- The least-significant bit indicates that the file synchronization pass has been completed (001).
- The next significant bit indicates that pass the metadata synchronization pass has been completed (010).
- The next significant bit indicates that the playlist synchronization pass has been completed (100).

For example, a value of 0 from this field means that no synchronization has completed, and a value of 5 (101) means that the file and playlist passes have been completed, but

that the metadata has not been completed. A value of 7 (111) indicates that all synchronization passes have been completed.

These flags are *not* cleared if the device is made unavailable. When a disk is moved out of the active slot while in a multi-disk changer, the disk is not made unavailable, only inactive. Therefore this action does not clear any existing synchronization flag values.

For detailed information about what the MME does at each synchronization pass and a complete list of synchronization events, see the section “Synchronization events” in the chapter MME Events.

The MME sets the synchronization flags when the synchronization process has finished inserting or updating the immediate contents of a folder. This behavior means that the client application can monitor the synchronization flags knowing that once the flag is set the contents for a mediastore folder in the MME database will not change.

## Nonblocking synchronization function calls

Synchronization function calls are all nonblocking; they leave the client application free to start media playback or perform other tasks.

This design means that the client application does not need to wait for synchronization to complete before it begins playing media for the end user. It can check for completion of the first synchronization pass and begin playing media while the MME synchronization process is updating its database with metadata and creating playlists.

## Pending synchronizations

If the MME receives a request to synchronize a mediastore but it does not have a thread available to perform the synchronization, it places the request in its “synchronization pending” queue until a thread becomes available.

Queued synchronizations can be canceled just like active synchronizations, by calling *mme_sync_cancel()*.

## Optimization of synchronization for starting playback on slow devices

To optimize system performance when starting playback on slow devices, the MME performs a foreground synchronization merge as soon as it has a file ID (*fid*) marked as a “first *fid*” (first playable) file. This action reduces the time required to start playback on slow devices.

## Ignoring specified file types

You can configure the MME synchronization process to:

- ignore certain file types, or files with specific strings in their names
- synchronize only specified file types

For efficient synchronization, you should configure your MME to skip files that begin with ".", and to synchronize only specified file types. For more information, see the chapter Configuring Media Synchronizations in the *MME Configuration Guide*.

The MME automatically always skips the files `.` (current directory) and `..` (parent directory) because they would cause recursion.

## Database clean up during synchronization

When it is completing synchronizations of a mediastore, the MME may attempt to cleanup unused references to metadata if:

- the mediastore being synchronized is prunable from the database, and
- the current synchronization is not the first synchronization of the mediastore

The cleanup proceeds as follows:

1. As the MME synchronizes the mediastore, it deletes from its database entries for files that are no longer on the mediastore.
2. When it has completed its last synchronization pass, the MME knows that it no longer requires metadata for these deleted entries and that it can delete references to their metadata.
3. The MME performs the cleanup, deleting unused references to metadata, after it delivers the MME_EVENT_MS_*PASSCOMPLETE event for the last requested synchronization pass.

The cleanup may:

- Take up to several seconds, depending on the size of the MME database, causing a corresponding delay between delivery of the MME_EVENT_MS_*PASSCOMPLETE event and delivery of the MME_EVENT_MS_SYNCCOMPLETE event.
- Cause the QDB to consume a large portion of CPU resources for the duration of the operation.

To prevent a track session "leak" — an accumulation of useless track sessions in the **`trksessions`** table — when a mediastore is pruned from your database, you should also delete from your database all track sessions that use tracks on that mediastore. For information about how to delete track sessions, see "Deleting track sessions" in the chapter Playing Media.

## Folder synchronization

The MME can be configured to deliver events when it starts and completes a folder synchronization. This capability may be used in association with prioritized folder synchronization, or as an alternative to polling a folder's *synced* column to monitor the

progress of its synchronization. See the chapterConfiguring Media Synchronizations in the *MME Configuration Guide* for information about configuring the MME to deliver folder synchronization events.

### Folder event sequence

The order of delivery of folder synchronization events for a specific folder is:

1. MME_EVENT_MS_SYNC_FOLDER_STARTED
2. MME_EVENT_MS_SYNC_FOLDER_COMPLETE
3. MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE, if recursive synchronization of the folder is requested

The MME delivers the MME_EVENT_MS_SYNC_FOLDER_STARTED event synchronously when it starts synchronization of a folder. At the same time, the MME queues the two other folder synchronization events for delivery, so that database changes associated with these two events are completed before the events are delivered.

This behavior means that it is normal for the client application to see MME_EVENT_MS_SYNC_FOLDER_STARTED events for child folders before it sees the MME_EVENT_MS_SYNC_FOLDER_COMPLETE event from the parent folder.

Because the events MME_EVENT_MS_SYNC_FOLDER_COMPLETE and MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE are queued, they consume one slot in the synchronization merge buffer space, if this space is used.

### Using `mme_folder_sync_data_t` information

All folder synchronization events use the structure **`mme_folder_sync_data_t`** to deliver information, but all synchronization events do not use all members of this structure. The different folder synchronization events deliver data in **`mme_folder_sync_data_t`** as follows:

MME_EVENT_MS_SYNC_FOLDER_STARTED

File pass: *num_files*=0; *num_folders*=0.
Metadata pass: *num_files*=0; *num_folders*=0.

MME_EVENT_MS_SYNC_FOLDER_COMPLETE — folder new or changed.

File pass: *num_files*= number of files in the folder; *num_folders*=number of child folders in the folder; *num_playlists*=number of playlists added to the **`playlist`** table.
Metadata pass: *num_files*=number of files updated; *num_folders*=0.

MME_EVENT_MS_SYNC_FOLDER_COMPLETE — folder not changed.

File pass: *num_files*=0; *num_folders*=0; *num_playlists*=0.
Metadata pass: *num_files*=number of files updated; *num_folders*=0; *num_playlists*=0.

MME_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE

File pass: *num_files*=0; *num_folders*=number of child folders synchronized. Metadata pass: *num_files*=0; *num_folders*=number of child folders synchronized.

See also the documentation for the individual events.

## Synchronizing playlists

Playlist synchronization converts playlist table entries into an ordered list of file IDs and places these file IDs in the **`playlistdata`** table.

Playlist synchronization includes the following behavior when finding a file in the MME database to match a playlist file:

- For every entry in the playlist, the MME attempts to match the filename for that entry in the playlist with a filename in its database.
- For playlists on MediaFS devices, the MME will search in its database for up to 100 matches of a filename in the playlist (same filename and same mediastore). If the database contains more than 100 matches, any matches above 100 are ignored.
- When the MME has completed its search, it associates with the playlist the file in the MME database that best matches the file in the playlist.

If a playlist file's timestamp or file size changes from what it was during the file synchronization pass, the synchronization process:

1. Removes the existing playlist file from the **`playlist`** table.
2. Creates a new entry for the file in the **`playlist`** table.

This new entry in the **`playlist`** table is *not* automatically synchronized. It requires a playlist synchronization pass to produce its ordered list of file IDs in the **`playlistdata`** table.

### Synchronizing a specific playlist

To synchronize a specific playlist (rather than synchronizing all playlists on a mediastore) call the function *mme_playlist_sync()*.

# Types of synchronization

The MME supports the following kinds of synchronization:

- full, recursive synchronization
- directed synchronization
- file synchronization

## Full, recursive synchronization

The default behavior for the MME is to automatically initiate full, recursive synchronization on detection of a new mediastore. With full, recursive synchronization, the MME scans all files on the mediastore and updates the MME database with all relevant information and metadata. To initiate full, recursive synchronization, call *mme_resync_mediastore()*.

## Directed synchronization

Directed synchronization synchronizes only the folders and files on 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 even at a later time.

To initiate directed synchronization, call *mme_sync_directed()*.

To improve the end user's ability to browse through a mediastore, such as an iPod, the MME makes available the MME_SYNC_OPTION_CANCEL_CURRENT flag. If the MME is performing a synchronization on a mediastore and the HMI needs to start a new directed synchronization (because, for example, the user has started browsing through a different folder), the HMI can use this flag when calling *mme_sync_directed()* to tell the MME to cancel the current synchronization and queue the new directed synchronization request for execution.

Directed synchronization is available only for mediastores with hierarchical directory structures: HDDs, iPods, USB sticks, data CDs, etc. It is not available for mediastores, such as music CDs, that have a single level directory structure.

### Directed synchronizations and missing folders

If a directed synchronization is unable to find on a mediastore a folder that is in the MME database, it deletes the folder and its contents from the MME database.

This behavior means that the client application can remove a folder from a mediastore, then use directed synchronization to remove this folder from the MME database.

## File synchronization

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 occurred: a file has been deleted, added, moved, or renamed.

File synchronization can be performed only with certain media store types. For example, this functionality it is not supported for use with iPods.

To initiate file synchronization, call *mme_sync_file()*.

# Updated database tables

The MME database tables listed below are updated during the synchronization process:

- File pass:
  - **folders**
  - **library**
  - **mediastores**
  - **playlists**
- Metadata pass:
  - **folders**
  - **library**
  - **library_***
- Playlist pass:
  - **folders**
  - **playlistdata**
  - **playlists**

## Media information and metadata

The MME **library** table provides a single view of metadata for different metadata formats (iTunes, Windows Media). If a metadata field is not supported in a file format, that field is simply left empty.

If the MME synchronization processes cannot find a title in a file, the MME sets the **library** *title* field to NULL. The client application can check a file's *title* field. If the field is set to NULL, it knows that the file does not have a decodable title, and it can handle the situation appropriately.

## Custom information and metadata

The MME has a customizable table, where you can add your own information tied to the MME library tables and access it as suits your needs. See the file **mme_custom.sql** for a sample schema.

For detailed information about when during the synchronization process specific tables are updated, see the section "Synchronization events" in the chapter MME Events.

# Working with synchronizations

The default configuration for the MME is to automatically detect mediastores and to automatically initiate their synchronization, updating the MME **library** and other tables with all relevant information and metadata. As a minimum, your client application should register to receive synchronization events in order to monitor the status and progress of synchronizations. You can also:

- instruct the MME to synchronize a mediastore by calling *mme_resync_mediastore()*
- find out if a mediastore has been synchronized, and if so, what passes have been completed, by calling *mme_sync_get_msid_status()*
- get the status of an synchronization in progress by calling *mme_sync_get_status()*
- call *mme_sync_cancel()* with the *msid* set to the mediastore ID to cancel synchronization of that mediastore, or with the *msid* set to **`0`** to cancel all current and pending synchronizations

For information about how to configure the MME's synchronization options, see the chapter Configuring Media Synchronizations in the *MME Configuration Guide*.

## Determining if resynchronization is needed

You can compare the *lastseen* and *last_sync* fields in the **`mediastores`** table to determine if you need to resynchronize a mediastore, and skip unnecessary resynchronizations. Both fields use the MME's internally derived time.

The *lastseen* field contains the time when the mediastore was last inserted into the MME, and the *last_sync* field contains the time of the mediastore's most recent synchronization. If the *lastseen* field is greater than the *last_sync* field, the mediastore may have changed since the last synchronization — it left the system and returned, and could have been changed: it should be resynchronized.

## Skipped synchronizations

The MME delivers the event MME_EVENT_SYNC_SKIPPED to indicate that it found a mediastore that could have been synchronized, but did not synchronize it for one of the following reasons:

- automatic synchronization is disabled; the client application must specifically request the synchronization
- automatic synchronization is enabled, but an internal event handler indicates that synchronization should not be done
- the mediastore is identified as a mediastore type that should not be automatically synchronized (e.g. an iPod)

## Setting a priority folder

The client application can instruct the MME to synchronize a specified folder first. You can use this feature to reduce the time required to make metadata available or start playback of media requested by the end user.

This capability can be useful when your client application is displaying the current view of synchronized directories during a synchronization process during startup, or when a new mediastore is inserted. If a user selects a displayed directory before the

MME has completed synchronization, your client application can set the selected directory as a priority folder. The MME will synchronize this directory first and the client application can update its display for the user as it receives MME_EVENT_MS_UPDATE events.

Call *mme_setpriorityfolder()* to tell the MME to pause any ongoing synchronizations and synchronize the specified folder before resuming the rest of the synchronization.

### Synchronization behavior with priority folders

The priority folder feature:

- supports one priority folder per media store being synchronized.
- silently ignores requests to synchronize:
  - the folder currently being synchronized
  - any folder below the current folder (because it will have already been synchronized)
- is *not* recursive: the MME will synchronize the only the priority folder before resuming its normal synchronization folder sequence
- if triggered during the *first* synchronization pass, completes all requested synchronization passes on the priority folder before resuming its normal synchronization folder sequence
- if triggered during a metadata or playlist synchronization pass, it completes the current synchronization pass only
- has no effect on a mediastore that is not currently being synchronized

If during a synchronization you set a priority folder, and you then set a new priority folder before synchronization of the first priority folder has completed, in most, but *not all* cases, the MME will:

1. Synchronize the most recently set priority folder.
2. Complete synchronization of the previously set priority folder.
3. Complete the general synchronization.

In the event that you attempt to set a new priority folder before the synchronization process has checked for the first priority folder you requested, the MME will drop the first priority folder request, and start synchronization with the newly requested priority folder. The first priority folder requested will be synchronized with the other folders in the general synchronization process.

## Removing file entries from the MME tables

You can instruct the MME to remove specific files from its database. To remove information for a specific file from the MME database, call *mme_sync_file()* with the *new_msid* set to 0 and *new_filename* set to NULL.

### Cleaning up the library after removing files

When the MME copies or rips a file, it places the file ID *fid* for the destination file in the *copied_id* field for the source file in the **`library`** table. If at a later time the destination file is deleted, this *copied_id* field becomes invalid, because it points to a file that is no longer in the MME library.

Checking the validity of *copied_id* fields is potentially a very costly (time-consuming) operation and is not performed by normal synchronizations. However, the *options* parameter for the synchronization functions *mme_sync_directed()* and *mme_resync_mediastore()* includes a flag (MME_SYNC_OPTION_CLR_INV_COPIED) that you can set to force the synchronization to check the validity of *copied_id* fields and set all invalid instances of this field to zero (file not copied).

To clean up invalid *copied_id* fields, call either *mme_sync_directed()* or *mme_resync_mediastore()* with the mask for the *options* parameter set to MME_SYNC_OPTION_CLR_INV_COPIED. With this option set, the synchronization operation will clean up invalid *copied_id* fields at the end of the file synchronization pass, if the following are true:

- The mediastore being synchronized has been synchronized before.
- The synchronization request has asked for at least the file synchronization pass.

**CAUTION:** Calling either *mme_sync_directed()* or *mme_resync_mediastore()* with MME_SYNC_OPTION_CLR_INV_COPIED set will clean up invalid *copied_id* fields for the *entire* MME database, not just for the library entries that correspond to the mediastore being synchronized. This operation can take a long time, and you should use it *only* after deleting from your database media files that were created by a copy or ripping operation (using the *mme_mediacopy_*()* functions).

## Repairing inconsistencies

If you encounter problems with a folder and its child items (files and subfolders) after a synchronization, you may be able to use *mme_sync_db_check()* to repair inconsistencies between the MME database and mediastores with POSIX compliant filesystems.

For more detailed information about checking for and correcting inconsistencies, see the information provided with the *mme_sync_db_check()* function in the *MME API Library Reference*.

### Determining if a file should be shown

The MME's file synchronization pass deletes from the MME database items no longer found on their mediastore *before* adding new items.

A client application may use the *seen* columns in MME tables to determine if a file on a mediastore can be shown to end-users. Only files marked as "seen" should be shown (files with their *seen* column set to **1**, meaning that the file synchronization pass found them on the mediastore).

Client applications should show end-users only files that have been found on a mediastore by the file synchronization pass, using logic based on timestamps: if the *last_sync* value of a **library** table entry for a file is greater than the *lastseen* time of the **mediastores** table entry for the mediastore that file is on, the file is on the system (because the entry has gone through a first synchronization pass since the mediastore was last placed in the system).

## Gracenote classical music support

The MME supports Gracenote classical music metadata in its library. To enable support for Gracenote classical music:

1. Enable Gracenote support in the configuration file **mme.conf** by setting the **<gracenote>** element to **true**.
2. Save the file and restart the MME.

For more information about how to make these changes, see "Gracenote support" in the *MME Configuration Guide* chapter Configuring Metadata Support.

The fields described in the table below are used in the MME library to support Gracenote classical music metadata. You should use the metadata in these fields primarily for display purposes, because at this time many classical music entries in the Gracenote database do not carry sufficiently precise metadata.

A single field in the MME **library** table may hold more than one instance and type of metadata, presented as comma separated strings. For example, when the *artist* field in the MME **library** table is built from Gracenote metadata, it can contain zero or more soloists followed by zero or one conductor; if, for instance, the field contains two names, it is not possible to deduce whether these two names are for two soloists or for a single soloist and a conductor.

In the table below, mandatory metadata is shown in square brackets: [*mandatory string*], and optional metadata is shown in curly braces: **{***optional string***}**.

| Field | Content |
|---|---|
| artist | 0 or more **{***soloist(s)***}**, 0 or 1 *conductor* |

*continued...*

| Field | Content |
|---|---|
| composer | composer short name |
| ensemble | 0 or 1 **{***ensemble***}**, 0 or 1 **{***choral ensemble***}** |
| opus | [*opus title*] **{**In *key***}**, **{***opus number***}**, **{***catalogue number***}**, **{***opus nickname***}** |
| title | [*movement number*], [*movement tempo or text title*] |

The remaining fields in the MME library hold their normal values.

For Gracenote classical music, the MME doesn't use the *soloist_id* or *conductor_id* fields, because the Gracenote metadata doesn't provide unique identifiers for these metadata.

# Chapter 6

# Playing Media

## In this chapter. . .

This chapter describes how to work with track sessions and play audio media on the MME.

# About playing media with the MME

The MME is designed to facilitate development of a user-friendly, efficient, and versatile HMI for playing diverse media. A client application can instruct the MME to begin playing media from a mediastore even before the mediastore has been synchronized with the MME database. However, most client applications will start synchronizing a mediastore before starting playback, so they can provide metadata, such as song artist and genre, to their end users.

To play media through the MME, the client application requires:

- a connection to an MME control context
- one or more mediastores with, in most cases, at least the first synchronization pass underway
- an appropriate output device connected to the control context
- an MME track session with playable tracks

- For more information about track sessions, see "Working with track sessions" below.
- For more information about playlists, see the chapter Playlists.
- For specific information about playing videos, see the chapter Playing and Managing Video and DVDs.
- iPods and Bluetooth devices require special consideration. For more information about how to work with track sessions and playback with these devices, see the chapters:
  - "Working with iPods"
  - "Working with Bluetooth Devices"

# Working with track sessions

A *track session* is the basic unit for playing media. It is created by an SQL query or an explorer API function that generates a list of media tracks that can be played in a control context. Each track in a track session is identified by its zero-based offset in the list; this method permits duplicate file IDs (*fid*s) in a track session.

To play media, the client application must:

**1** Create a track session by calling *mme_newtrksession()*. This function delivers the track session ID, which the client application can use to:

- set the track session as the current track session for the control context
- remove the track session from the MME database

2. For track sessions that use media from unsynchronized mediastores (file-based track sessions), use the explorer API to discover and add track sto the track session. For more information, see the chapter Unsynchronized Media in this guide. This step is not needed for track sessions that use media from synchronized mediastores (library-based track sessions), because for these types of track sessions, the call to *mme_newtrksession()* populates the track session with the tracks to be played.
3. Set the track session by calling *mme_settrksession()*. Once a track session is set, the client application can begin playback or perform other operations, such as fast-forwarding, setting the random or repeat mode, and so on
4. Start playback, by calling *mme_play()*, or another function.

- The MME supports multiple track sessions; to determine which is the current track session, call *mme_trksession_get_info()*.
- 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 by calling *mme_settrksession()* with *trksessionid* set to 0 (zero).

For more information about the fields in the track session table, see the **`trksessions`** table entry in the appendix: MME Database Schema Reference of the *MME API Library Reference*.

## Types of track sessions

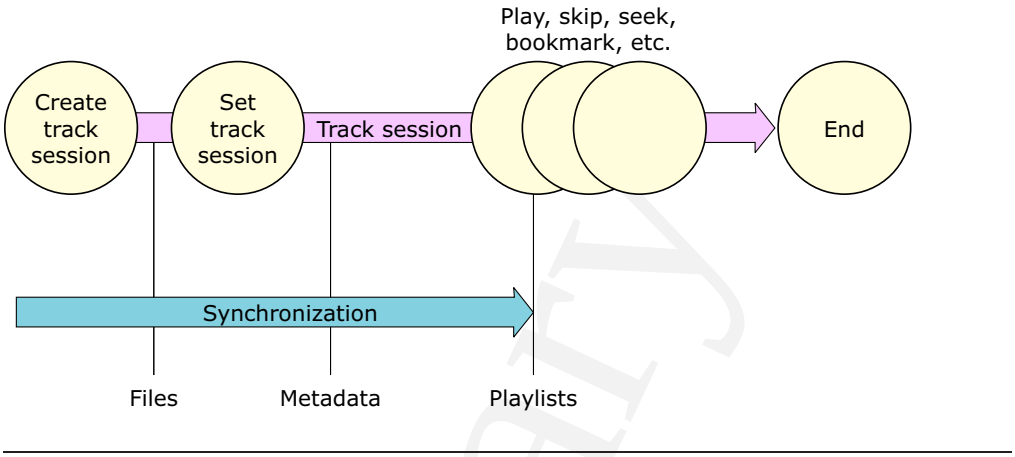
The MME supports two types of track sessions:

- library-based
- file-based

### Library-based track sessions

A library-based track session is built with files from mediastores that have been synchronized and, therefore, have entries in the **`library`** table in the MME database. To create a library-based track session, simply proceed as with previous releases, calling *mme_newtrksession()* with the *mode* argument set to MME_PLAYMODE_LIBRARY (0).

The figure below illustrates the flow of activities from the creation of a track session to the end of the track session, with a mediastore synchronization in the background.




*Playing media with the MME.*

#### File-based track sessions

A file-based track session is a track session built with files discovered through the the MME's explorer API. Unlike tracks in a library-based track session, tracks in a file-based track session can be from unsynchronized mediastores and do *not*, therefore, have to have entries in the **`library`** table. For more information about the MME's explorer API, see the chapter Unsynchronized Media in this guide.

### Track sessions and playlists

The difference between a track session and a *playlist*, is that a playlist is an arbitrary collection of tracks that is created by a user, either by selecting individual songs or by creating some selection criteria (such as all the songs by a particular artist), represented as an SQL statement. The user can name a playlist and store it. A playlist isn't associated with a control context.

To play a playlist, the client application creates a track session from the playlist and associates that track session with a control context. Set the track session with the playlist to be the current track session by calling *mme_settrksession()*. For more information about playlists, see the chapter Playlists.

## Creating track sessions

Your application should create either a library-based track session or a file-based track session, depending the synchronization status of the mediastore with the media to be played:

- for synchronized mediastores, create a library-based track session
- for *un*synchronized mediastores, create a file-based track session

MME track sessions support duplicate file IDs (*fid*s), because the MME references tracks in track sessions by their offsets, not their *fid*s. To start playback with a specific track offset in a track session, use *mme_play_offset()*.

## Creating library-based track sessions

Library-based track sessions are used for playing found on synchronized mediastores, and are the most commonly used type of track session.

To create a library-based new track session, call *mme_newtrksession()* with an SQL statement to select the tracks you want to play and the *mode* argument set to MME_PLAYMODE_LIBRARY (0). This function will use your query statement to:

- retrieve the tracks you want to play from the MME database
- create a track session in the **`trksessions`** table

**CAUTION:** Always use MME functions to update the **`trksessions`** table. *Never* write directly to this table. If you write directly to this table, you will make its data unreliable.

Below are some examples of how your client application could create a library-based track session and play it. To keep things simple, these examples assume that you have already connected to the MME database and MME resource manager, and they don't check return codes, which your application should do.

### Playing all tracks in the MME library

We can start with the simplest case: create a track session to play all the tracks in the library. This case has the following steps:

1. Create a track session that includes all audio tracks from all available mediastores in the library.
2. Set this track session as the active track session in the current control context.
3. Play all tracks in the track session, in the alphabetical order of the titles.

Since we are not interested in metadata or playlists, we can start playing tracks from any new mediastores after only the first synchronization pass has completed on these stores.

```
mme_hdl_t   *mme;
char        *sql;
uint64_t    trksessionid;

mme = mme_connect("/dev/mme/default", 0);        // Connect to MME
sql =   "SELECT library.fid AS fid FROM library "
        " INNER JOIN mediastores ON mediastores.msid = library.msid "
        " WHERE mediastores.available=1 AND ftype=1 "
        " ORDER BY title";
```

```
// create the new track session
mme_newtrksession( mme, sql, MME_PLAYMODE_LIBRARY, &trksessionid );

// set the new track session as the active track session
mme_settrksession( mme, trksessionid );

// start playing the track session,
// pass in a fid of 0 to start from the beginning.
mme_play(mme, 0);
```

Note that:

- The SELECT statement you pass to *mme_newtrkssession()* must not have a final semicolon. The final semicolon must be omitted, because this statement is in fact a sub-statement; *mme_newtrksession()* places this sub-statement into a larger SELECT statement.
- The query requests only audio tracks (*ftype*=1).
- The result for the statement you pass to *mme_newtrksession() must* include a *fid* column. In fact, the MME disregards any other column, so it is most efficient to just select *fid*.

For more information about the SQL queries used with the MME and QDB as well as recommendations, see the chapter Working with the MME Database and SQL.

#### Excluding mediastore *fid*s from track sessions

The MME adds a *fid* to the **`library`** table for many types of mediastores, including iPods and type MME_STORAGETYPE_DEVB mediastores (which includes HDDs, USB sticks, CDs and DVDs).

When composing queries for a track session, you should exclude *fid*s that refer to mediastores by adding a **`WHERE`** clause to the query statement to select the file type entry (*ftype*) you need. For example to select only entries where *ftype*=FTYPE_AUDIO:

```
SELECT fid, ftype, title FROM library WHERE ftype=1 ORDER BY title;
```

### Creating and modifying file-based track sessions

File-based track sessions are used for playing found on unsynchronized mediastores.

To create a file-based track session:

1. Use the MME's explore API functions (*mme_explore_*()*, etc.) to explore a mediastore and retrieve information about tracks of interest on the mediastore.
2. Create a file-based track session by calling *mme_newtrksession()* with the *mode* argument set to MME_PLAYMODE_FILE (1).
3. Set the track sesssion by calling *mme_settrksession()*.

4. Call one of *mme_trksession_append_files()* or *mme_trksession_set_files()* to add files to the track session.
5. Proceed with playback and other functionality as with library-based track sessions.

You don't need to use the explorer API before you create or set the track session. You can use it at any time to discover tracks of interest, which you can then add to your track session using one of the methods described below.

#### Modifying a file-based track session

You can explore mediastores and add tracks to your track session at any time after creating the track session, or after you have started playback. You can change an existing file-based track session by:

- calling *mme_trksession_append_files()* to add newly explored files to the track session
- calling *mme_trksession_set_files()* to replace all the tracks in the track session with a new list of tracks

## Setting track sessions

After you have created a track session, you must set it by calling *mme_settrksession()*. When you call *mme_settrksession()*, the MME takes a snapshot of the SQL statement that represents the track session and stores it in the **`trksessionview`** table. This entry in the **`trksessionview`** table changes only when a new track session is set, or if you call the function *mme_trksessionview_update()*.

### Setting a track session before synchronization has completed

If a client application sets a track session before the MME has completed the first synchronization pass of a mediastore (before it receives the event MME_EVENT_MS_1PASSCOMPLETE), the track session contains only a subset of the data available on the mediastore. For example, if the client application calls *mme_settrksession()* before it receives the event MME_EVENT_MS_1PASSCOMPLETE, the track session it sets will contain only those tracks that were synchronized up to that point; the remaining tracks on the mediastore will *not* be included in the track session.

**CAUTION:** If the SQL **`SELECT`** statement used to create the track session uses metadata, the client application has to wait for the event MME_EVENT_MS_2PASSCOMPLETE before making its final update to the **`trksessionview`** table. For example, **`SELECT fid FROM library WHERE artist_id=4 AND msid=3`** uses metadata, so all *fid*s for the track session will not be found until after the second synchronization pass is complete. However, a query such as **`SELECT fid FROM library WHERE msid=3 ORDER BY filename`** uses only information provided by the first synchronization pass, so all *fid*s for the track session are found by the first synchronization pass.

To enable playback as quickly as possible while ensuring that all requested tracks are included in the track session, the client application should update the **`trksessionview`** table when it knows that the first syncrhonization pass has completed, and, for large mediastores that can take a long time to synchronized fully, periodically so that the track session does not run out of tracks. For example, with a large mediastore with 10,000 files, the client application can do something like the following, assuming that synchronization was started automatically, and that the SQL **`SELECT`** statement does not use metadata:

1. Create the track session (*mme_newtrksession()*).
2. MME_EVENT_MS_SYNCFIRSTFID received: set the track session (*mme_settrksession()*) with one *fid*.
3. Start playback (*mme_play()*, then *mme_resume_state()*).
4. Refresh the data in the **`trksessionview`** table (*mme_trksessionview_update()*).
5. If MME_EVENT_TRKSESSIONVIEW_INVALID is received, loop until MME_EVENT_TRKSESSIONVIEW_UPDATE is received, indicating that the MME has begun updating the **`trksessionview`** table in the background.
6. At one minute intervals, update the data in the **`trksessionview`** table (*mme_trksessionview_update()*) until the event MME_EVENT_MS_1PASSCOMPLETE is received. When this event is received, refresh the **`trksessionview`** table one last time.

- Updates of the **`trksessionview`** table may take several seconds. It is best to keep these to a minimum. Full playback capabilities are available while the MME performs these updates.
- The MME writes blocks of entries to the **`trksessionview`** table in the background. The size of this block is configurable. See "Setting the number of tracks written to the trksessionview table" in the *MME Configuration Guide*. When it finishes writing a block of entries, the MME delivers the event MME_EVENT_TRKSESSIONVIEW_UPDATE. When it finishes writing entries for all available tracks (tracks that have been synchronized thus far), the MME delivers the event MME_EVENT_TRKSESSIONVIEW_COMPLETE.
- Delivery of the event MME_EVENT_TRKSESSIONVIEW_COMPLETE means only that there are no more entries for tracks to be written to the **`trksessionview`** table. When synchronization completes its first pass, you will need to call *mme_trksessionview_update()* again to update the **`trksessionview`** table.
- Metadata for all tracks in the track session is not available until synchronization has completed its second pass and delivered the event MME_EVENT_MS_2PASSCOMPLETE.

## Clearing track sessions

You can clear a track session by:

1. calling *mme_stop()* to stop the track session
2. calling:
   - for library-based track sessions: *mme_settrksession()* with *trksessionid* set to 0 (zero)
   - for file-based track sessions: *mme_trksession_clear_files()*

## Deleting track sessions

To prevent a track session "leak" — an accumulation of useless track sessions in the **`trksessions`** table — you should delete track sessions from the **`trksessions`** table in the following circumstances:

- When you prune from the MME database the mediastore with the tracks used by the track session.
- When the number of track session in the **`trksessions`** table increase above a threshold that you define as optimal for your system and users.
- When a track session is older than a period you define as optimal for your system and users.
- When requested to do so by the user.

To delete a track session:

1. Call *mme_stop()* to stop playback on the track session.
2. Call *mme_settrksession()* with the *trksessionid* set to 0 (zero) to clear the track session.
3. Call *mme_rmtrksession()* to delete the track session.

# Monitoring and managing playback

After you have started playing a track session, you need to monitor the progress of the playback by checking the MME playback events. The example below shows a client application displaying messages on receipt of some MME events:

```
switch (msg.single.type) {
    case MME_EVENT_NONE:
        fprintf(stderr, "Received MME_EVENT_NONE (%d)\n",  MME_EVENT_NONE);
        break;
    case MME_EVENT_TIME:
        fprintf(stderr, "Received MME_EVENT_TIME (%d)\n",  MME_EVENT_TIME);
        break;
    case MME_EVENT_FILECHANGE:
        fprintf(stderr, "Received MME_EVENT_FILECHANGE (%d)\n",  MME_EVENT_FILECHANGE);
        break;
    case MME_EVENT_PLAYLIST:
        fprintf(stderr, "Received MME_EVENT_PLAYLIST (%d)\n",  MME_EVENT_PLAYLIST);
        break;
    ...

    default:
        fprintf(stderr, "Unknown Event Received (%d)\n",  msg.single.type);
        break;
}
```

## Setting the playback notification interval

While the MME is playing a track session, it delivers the event MME_EVENT_TIME at set intervals to notify the client application of playback progress.

The default interval between deliveries of MME_EVENT_TIME is 100 milliseconds, but the MME allows you to change this interval by calling *mme_set_notification_interval()* with the *time* set to the desired interval, in milliseconds.

Note that:

- The interval between deliveries of MME_EVENT_TIME remains constant regardless of the speed of the playback.
- The frequency of updates during playback depends upon the frequency of updates from audio drivers and devices. Depending on the frequency of updates received from audio drivers and devices, client applications may notice jitter in the reporting

of playback positions. For more detailed information, see *mme_set_notification_interval()* in the chapter MME API.

You should monitor for events indicating errors, changes in tracks (so you can display the metadata display, for example), completion of playback, etc. For a complete list of playback event types, see “Playback events” in the chapter MME Events.

You can also check on the progress of the playback by calling *mme_play_get_status()*. This function retrieves the status of playback, providing the total play time of the track and the play time elapsed.

## Knowing when playback has ended

In most environments, users want to continue playback without interruption until they explicitly request a change. Therefore, once the MME has started playback of a track session, it continues playback until:

- the client application issues instructions to the MME

  or:

- playback encounters an error that forces it to stop playback

  or:

- playback reaches the end of the track sessions

In short, once playback of a track session has started, the client application doesn’t need to do anything except monitor MME events, and pass information and metadata to the end user, until it receives one of these events:

- MME_EVENT_FINISHED — playback has stopped because there are no more tracks to play in the track session
- MME_EVENT_FINISHED_WITH_ERROR — playback has stopped due to an error

Both these events indicate that playback has stopped, and that action by the client application is required for playback to resume. No other events (not even MME_PLAY_ERROR_* events) require action from the client application for playback to continue.

If the client application instructs the MME to stop playback (e.g. by calling *mme_stop()*), the MME does *not* deliver an MME_EVENT_FINISHED_* event.

## Using random and repeat modes

The MME can set the playback mode of a library-based track session to play through the track session sequentially, repeat the track being played, repeat the entire track session, or play through the track session in random order.

Track sessions inherit their repeat and random modes from the control context in which they are created. Use the functions *mme_getrepeat()* and *mme_setrepeat()*, and *mme_getrandom()* and *mme_setrandom()* to get and set these modes.

- 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.
- iPods maintain their own random and repeat modes, which the MME can detect and set. For more information, see "Using random and repeat modes on iPods" in the chapter Working with iPods.

### Repeat and random modes with file-based track sessions

The explorer API and file-based track sessions are designed to allow the client application to manage its track sessions, discovering tracks and adding new tracks to a track session a few at time, as required by the end-user.

When new tracks are added to a track session in random mode, the new tracks are appended in pseudo-random order to the end of the track session; they are not integrated into the pseudo-random order for the entire track session. For example, if five tracks are added to a track session with 20 tracks, the order for the tracks in positions 0 to 19 remains the same, and the new tracks are appended in pseudo-random order in postions 20 to 24.

When playing tracks in a file-based track session, to change the next track that will be played without interrupting the track currently being played, the client application can call *mme_trksession_set_files()* with the *offset* parameter set to the required track.

## Starting playback from a specific track

The MME lets you start playback with a specific track from a track session. The method for starting playback from a specific track is different for library-based and file-based track sessions.

The MME can play a track that isn't in the current track session, as long as it has an active track session in the current control context. For more information, see *mme_play()*.

### Library-based track sessions

To start playback with a specific track in a library-based track session, instead of passing *mme_play()* a 0 (zero) for the *fid* argument, which starts playback with the first track in the track session, pass it the *fid* of the track you want to play. The MME will start playback with the track you specified, then continue playing the track session as determined by the position of the track in the track session and the random and repeat mode settings for the control context. For example:

- If repeat and random mode are off and the track you request is the second one in the track session, the MME will start with that track and play all tracks to the end of the track session. If the track you request is the last one in the track session, the

MME will play only that track, then stop playback, because it will have reached the end of the track session.

- If the repeat mode is turned on, the MME will start playing the requested track, then continue playing, repeating either the track or the entire track session, as determined by the repeat setting.
- If the random mode is turned on, the MME will start playing the requested track, then continue playing tracks as listed in its pseudo-random list.

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*.

### File-based track sessions

To start playback from a specific point in the file-based track session, use *mme_play_offset()*, passing it the zero-based offset of the track where you want to start playback. For example, to start playback with the 17th track in the track session set the *mme_play_offset()* function's *offset* argument to 16.

### Playing a track not in the current track session

The MME can play a track that isn't in the current track session, as long as it has an active track session in the current control context. For more information, see *mme_play()*.

## Pausing playback

Your client application should pause playback only in situations when it expects to resume playback or tear down the track session after a brief interval, such as when the end user sends a "pause" button command (MM_BUTTON_PAUSE) through the HMI. For situations when you expect a change to the system, such as a shutdown or a change to the mediastore being played, you should save the track session state and stop the track session. For more information, see "Stopping and resuming playback" below.

To pause playback temporarily, call *mme_play_set_speed()* with the *speed* set to 0. This action pauses playback until you call *mme_play_set_speed()* again with the *speed* set to something other than 0 (zero). Normal playback speed is 1000. For more information about how to use *mme_play_set_speed()*, see "Using fast forward and reverse" below.

## Stopping and resuming playback

Your client application should save the track session state and stop the track session when it expects or encounters a system change, such as a shutdown or a change to a mediastore. Such situations include, but are not limited to, the following:

- a CD change in a system with a CD changer
- removal of a device, such as an iPod

- user-initiated change to another activity, such as switching from playback to the radio tuner
- a system shutdown
- a shutdown of a supporting system, such as, for an MME system installed in an automobile, the automobile's being turned off

The operations required to stop then resume playback are a function of the capabilities of the device with the media tracks for the track session. These devices can be:

- devices, such as USB media stores or CD changers, that do not manage their own track sessions and, therefore, cannot save the state of a track session. For more information, see "Resuming playback" below.
- devices, such as iPods, that manage their own track sessions and, therefore, save the state of these track sessions. For more information, see "Resuming playback on iPods" in the chapter Working with iPods.

For information about resuming playback when using the explorer API, see "Pausing and resuming playback in a file-based track session" below.

### Resuming playback

If you want to stop, then resume playback of a track session on a device or devices that don't manage their own track session, you must:

**1** Stop the track session:

**1a** Call *mme_play_set_speed()* with the *speed* set to 0 (zero) to pause the track session.

**1b** Call *mme_trksession_save_state()* to save the play position and other information, such as the track session ID, about the track session.

**1c** If you plan on using *mme_play_resume_msid()* to resume playback, call *mme_set_msid_resume_trksession()* to set the mediastore ID to be used by *mme_set_resume_msid()*.

**1d** Stop the track session by calling *mme_stop()*.

**2** Do something else, such as switch to another activity or shut down the system.

**3** Resume the track session:

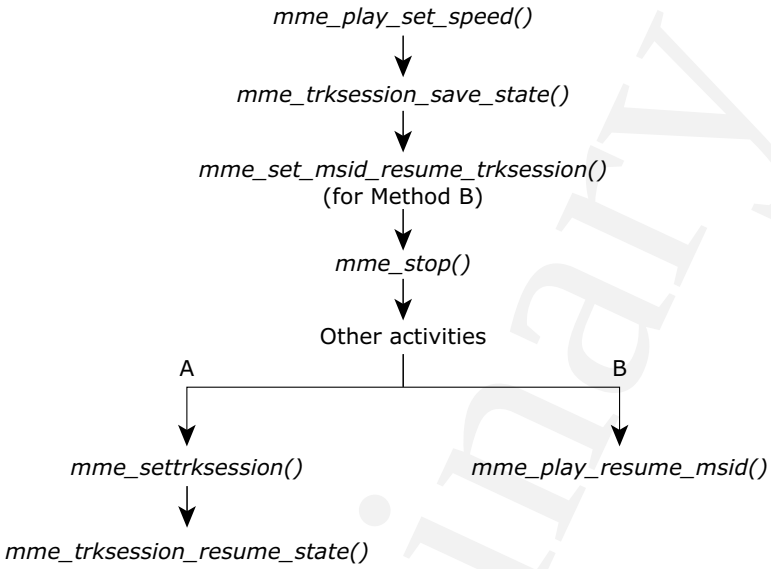
Method A (works only for devices and mediastores that don't support their own track sessions):

**3a** Call *mme_settrksession()* to reset the track session.

**3b** Call *mme_trksession_resume_state()* to resume playback of the track session.

Method B (works for all devices and mediastores):

**3a** Call *mme_play_resume_msid()* to resume playback of the track session (equivalent to calling *mme_settrksession()*, then *mme_trksession_resume_state()*).




*Stopping and resuming an MME-controlled track session.*

The strategies described below allow your client application to call *mme_trksession_resume_state()* to resume playback at a later time at exactly the position where it was stopped.

Saving the track session state with *mme_trksession_save_state() before* stopping it assures that, when you resume playback, the MME has all the information it needs to start playing the correct track at the correct position.

Pausing the track session before saving its state ensures that when playback resumes, it will be *exactly* at the correct position in the track: the user will not hear the last few milliseconds of music played if there is a delay between the call to *mme_trksession_save_state()* and the call to *mme_stop()*. If you do not pause the track session before saving its state, a situation like the following may occur:

- The playback is *n* milliseconds into a track.
- The client application calls *mme_trksession_save_state()*, then *mme_stop()*.
- There is a delay of *x* milliseconds between the call to *mme_trksession_save_state()* and the call to *mme_stop()*, which results in the state's being saved as *n* milliseconds, but playback stopping at *n* + *x* milliseconds.
- A call to *mme_trksession_resume_state()* starts playing the track at its saved state, so the end user hears the portion of the track from *n* milliseconds to (*n* + *x*) milliseconds a second time.

- Saving the track session state with *mme_settrksession()* and resuming it by calling *mme_set_msid_resume_trksession()* is intended for handling situations such as system shutdowns or mediastore changes. You can *not* save the track session state, continue using the track session, then attempt to resume it.

  In short, once you have saved the track session state you must stop using that track session until you are ready to resume it. If you want to mark a place in a track session, continue playback or do some other activity, then resume playback at the point you marked, use the MME's bookmark functions. For more information, see "Bookmarking tracks" below.

- Calling either *mme_settrksession()* or *mme_set_msid_resume_trksession()* regenerates the list of tracks used by the MME for playback in random mode (the entries in the *randomid* field of the **`trksessionview`** table).

## Pausing and resuming playback in a file-based track session

Tracks in a file-based track session do not have corresponding entries in the MME **`library`** table. This particularity means that when stopping and resuming playback, the client application must take care of saving and restoring the file name and current time position for the track to be paused and resumed.

### Pause playback in a file-base track session

To pause playback of a track in a file-based track session:

1. Call *mme_play_set_speed()* with the *speed* set to 0 (zero) to pause playback.
2. Save in a safe location;
   - the full filename for the paused track (without the mountpath)
   - the current time position for the track

### Resume playback in a file-base track session

To resume playback of a track in a file-based track session, assuming that the track was passed as describe in "Pause playback in a file-base track session" above:

1. Call *mme_newtrksession()* to create a new track session by with a device file ID (*fid*) for the mediastore with the track to be resumed.
2. Call *mme_settrksession()* to set the track session.
3. Call *mme_setautopause()* with the *enable* argument set to **`true`** to turn autopause on for the MME; with autopause enabled in the MME, when a track is played it begins in paused mode and remains paused until *mme_play_set_speed()* is called with the *speed* argument set to non-zero.
4. Call *mme_trksession_append_files()*, with the *filename* argument pointing to the filename you saved when you paused playback, to add the paused track to the new track session so that it can be played.

5. If your client application's connection is *not* O_SYNC; that is, if the MME does not completely execute requests before returning to the client, wait for the event MME_EVENT_PLAYAUTOPAUSED.
6. Call *mme_seektotime()* with the *time* argument set to the current time position you saved before pausing the track.
7. Call *mme_play_set_speed()* with the *speed* argument set to 1000 to resume playback at normal speed.

## Using fast forward and reverse

Use *mme_play_get_speed()* to get the current playback speed of a track session (expressed in units of 1/1000 of normal speed). Use *mme_play_set_speed()* to pause, reverse, and playback more slowly or more rapidly than normal playback speed. Set the *speed* argument as shown in the table below:

| Setting | Action |
|---|---|
| **<** 0 | Reverse at *speed* |
| = 0 | Pause playback |
| **>** 0 and **<** 1000 | Slow playback at *speed* |
| = 1000 | Normal playback |
| **>** 1000 | Fast playback at *speed* |

### Implementation note about fast-forward and fast-backward speeds

When setting fast-forward or fast-backward speeds, 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()*.

Some devices, such as iPods, do not maintain a constant fast-forward or fast-backward speed, but increase the speed according to the amount of time the fast-forward or fast-backward is maintained.

For devices with this behavior, there is no value in attempting to measure play time during a fast-forward or fast-backward. If you are testing with these devices, you can only ensure that fast-forward and fast-backward arrive at the end or beginning of a track faster with normal speed.

#### Fast-forward and fast-backward between tracks

Note that behavior when fast-forward or fast-backward move to a new track is:

- Device dependent — some devices, such as iPods, automatically set the speed to normal playback speed.

- Configurable by setting the **<AtEndOfSeek>** element, if supported by the device. The default setting is to continue seeking (fast forward of reverse). See "Configuring playback" in the *MME Configuration Guide*.

## Using seek to time, play at offset, and scan

To seek to a specific time in a track that is being played, use *mme_seektotime()*, passing it the time location to which you want to go and continue playback. For example, to skip ahead 15 seconds from the current position in a track, use *mme_play_get_status()* to get the current time location of the playback, then call *mme_seektotime()* with the current time location plus 15000.

Scan mode plays a track for a specified number of seconds, then moves to the next track, scanning all tracks in a track session. To use scan mode and set the time that the MME plays a track before moving to the next track, call *mme_setscanmode()* with the number of milliseconds you want to play each track before moving the scan to the next track. To get the current scan mode setting, use *mme_getscanmode()*. To turn scan mode off, call *mme_setscanmode()* with the time-to-scan argument set to 0 (zero).

## Gapless playback

When gapless playback is enabled, if two tracks on the same mediastore use the same graph type, when moving from one track to the other, the MME attempts to minimize the silence between tracks.

To enable gapless playback, you need to start **io-media** with the appropriate arguments. For more information, see "Configuring gapless playback" in the *MME Configuration Guide*.

## Viewing "previous" and "next" tracks

The MME stores information about tracks that have been played and will play in the track session in the **trksessionview** table. This information allows your client application to have the MME to move backwards through a track session, even if random play mode is enabled.

If random mode is off, playback advances through the tracks as they are listed in the *sequentialid* column of the **trksessionview** table. If random mode is on, playback advances through the tracks as they are listed in the *randomid* column of the **trksessionview** table. To view the previous or next tracks in a track session, use the file IDs listed before or after the current track in the relevant column.

## Using play frequency statistics

The MME maintains information about how many times a track has been played. This count includes fast forwards through the track. The client application can use this information to build a most-popular or top-50 list.

The number of times a track has been played is maintained in the *fullplay_count* field of the **library** table.

## Bookmarking tracks

The MME provides bookmarking capabilities that a client HMI can offer to end users. Bookmarks allow users to mark time locations on tracks in a track session and to resume playback from these locations. Bookmarks are recorded in the **`bookmarks`** table, and are identified by a bookmark ID (*bookmarkid*) and bookmark name (*name*), a mediastore ID (*msid*), and a track file ID (*fid*).

To view available bookmarks, query the **`bookmarks`** table. For example, to view all bookmarks for tracks on the current mediastore, where *current_msid* is the mediastore ID :

```
.
SELECT bookmarkid, fid, msid, name FROM bookmarks
    WHERE msid=current_msid ORDER BY name;
```

Use the functions *mme_bookmark_create()* and *mme_bookmark_delete()* to create and delete bookmarks, and *mme_play_bookmark()* to start playback from a specified bookmark.

# Managing track sessions during playback

This section describes how to manage track sessions during playback.

## Managing track changes across multiple mediastores

The MME performs track changes across different mediastores. This feature is supported by buffer level reporting. The MME provides the amount of time (milliseconds of playback) remaining in the MME buffer. Client applications can retrieve this information by calling the function *mme_play_get_status()*. This information gives device controllers a more accurate measure on which to base decisions to wake up devices, such as the system HDD.

### Aborting blocking reads

Client applications can use the time left in the MME buffer to decide to abort blocking reads in order to skip to tracks on a mediastore other than the HDD. Aborting a blocking read allows the MME to fulfill a request to start playback of a track on another mediastore, such as a CD, immediately. It does not have to wait for its buffer to empty, or for the device controller to wake up the HDD.

## Managing track sessions when a mediastore is removed

When a mediastore is removed from the system, the MME delivers the event MME_EVENT_MS_STATECHANGE, which carries the new state of the mediastore in **`mme_event_data_t.ms_state_change.new_state`**. If media on the mediastore is included in a currently playing track session, you can manage the change according to whether the track being played has been removed or is still on the system.

For example, if your track session includes tracks from both **`/fs/usb0`** and **`/fs/usb1`**, and the track being played is on **`/fs/usb0`**:

- If the user removes **`/fs/usb0`**, then playback is interrupted, but you can refresh the track session to include only tracks on **`/fs/usb1`** and resume playback.
- If the user removes **`/fs/usb1`**, then playback continues, and you can refresh the track session to include only tracks on **`/fs/usb0`** and continue playback of available tracks without interruption.

### Managing playback when the current mediastore is removed

If the track currently being played is on the removed mediastore, the MME delivers the event MME_PLAY_ERROR_DEVICEREMOVED, and stops playback:

- If the track session contained only tracks on the removed mediastore, you can inform the user that playback has stopped because the mediastore was removed, and request input.
- If the track session contained tracks from several mediastores, you can call *mme_trksessionview_update()* to update the track session (remove unavailable tracks), then call *mme_next()* or *mme_play()* to resume playback from a mediastore still in the system.

### Managing the track session if playback is not on the removed mediastore

If the track currently being played is *not* on the removed mediastore, the MME continues playback, and you can ensure that playback will continue uninterrupted to the end of the track session:

- If the track session includes tracks on the removed mediastore, call *mme_trksessionview_update()* to update the track session (remove unavailable tracks). When the currently playing track finishes playing, the MME will simply advance playback to the next track in the **`trksessionview`** table.

## Switching playback to another track session

The MME supports seamless switching between track sessions; that is, changing playback from one track session to another without interrupting playback, *if* the new track session includes the track currently being played.

To seamlessly change track sessions during playback:

1. Create a track session that includes the *fid* of the currently playing track.
2. Call *mme_settrksession()* to set the new track session.

The MME will:

- continue playback of the currently playing track
- when playback of this track is complete, continue playing tracks from the newly set track session

- 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).

For more information about MME behavior when switching playback between track sessions, see *mme_settrksession()* in the *MME API Library Reference*.

# Chapter 7

# Playlists

## *In this chapter. . .*

Playlists can come from two sources:

- playlists included with mediastores
- playlists created by the client application, through the MME

This chapter describes how to work with playlists.

For information about synchronizing playlists, see "Synchronizing playlists" in the chapter Synchronizing Media.

# Creating track sessions from playlists

The example below shows how to create a track session from a playlist. The MME synchronizes playlists in the playlist synchronization pass, and stores them in the **`playlists`** table. If we have a playlist called "My Playlist" that was built from a **`SELECT`** statement, we can:

1. Use the QDB function *qdb_statement()* to get the SELECT statement for the playlist.
2. Pass the statement to *mme_newtrksession()* to create a track session with the same tracks as the playlist.
3. Set the new track session.
4. Play the tracks in the track session, in sequence from the beginning.

```
// Run the SQL statement.
qdb_statement(&db,
    "SELECT statement FROM playlists WHERE plid=%lld;",
    plid )
res = qdb_getresult( &db );

// Create a new track session from the result.
mme_newtrksession( &mmehdl, (char*)qdb_cell(res, 0, 0),
    MME_PLAYMODE_LIBRARY, &trksid );


// Set the new track session as the active track session.
mme_settrksession( &mmehdl, trksid );

// Start playing the track session from the beginning,
// passing in a fid of 0 to start from the beginning.
mme_play(&mmehdl, 0);
```

## Excluding missing playlist files from track sessions

When it synchronizes playlists, the MME inserts in the **`playlistdata`** table a *fid* of 0 (zero) for any files that it cannot find. This action creates a record of files in a playlist that are not found, and causes the MME to check for the existence of these files when it performs subsequent resynchronizations.

If you build a track session using the **`playlistdata`** table, you should explicitly exclude files with a *fid* of 0 by adding the clause **`WHERE fid != 0`** to track session queries made to the **`playlistdata`** table.

## Combining playlists into a track session

The MME supports multiple instances of the same file ID (*fid*) in a track session, so you can combine playlists with duplicate *fid*s and have the MME play all the tracks in the combined playlists.

# Examining playlists

The MME includes functions that allow you to open and examine a playlist file. To open a playlist file and examine the contents of the playlist:

1. Call *mme_playlist_open()* to:
   - create a playlist session connection handle (**`mme_playlist_hdl_t`**)
   - open a playlist examination session
2. Call *mme_playlist_items_count_get()* to get the number of items in the playlist you are examining.
3. As required, use *mme_playlist_position_set()* to move to a specific entry in the open playlist.
4. Call *mme_playlist_item_get()* to retrieve an item from the current position in the playlist, setting the *flags* argument as required to convert the playlist entry into a file.
5. Call *mme_playlist_close()* to close the playliste examination session and free up its buffers.

- *mme_playlist_open()* 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, this function fails.
- The playlist examination API is very similar to the explorer API. How to use this API is described in the chapter Unsynchronized Media.

#### Case-sensitivity in playlists

Playlists are case-sensitive. Case-sensitivity applies to:

- playlist names
- pathnames to playlists
- paths inside the playlist files
- playlist entries

For example **`songs/my_playlist`** is *not* equivalent to **`songs/My_playlist`**. Similarly, in an M3U file **`songs/Dark Side of the Moon.mp3`** is *not* equivalent to **`Songs/Dark Side of the moon.mp3`**.

# Creating playlists

The MME supports user-created playlists. Please note the following about user-created playlists:

- Client applications must manage their own playlists, placing the file IDs (*fid*s) of tracks in the **`playlistdata_custom`** table.
- Resynchronizing a mediastore does not affect user playlists.
- Pruning a mediastore deletes user playlists. You should add a trigger to remove entries in the **`playlistdata_custom`** table when a mediastore is pruned, because ift he mediastore is re-inserted into the system, its files will get new file IDs that will not correspond to the file IDs in the playlists
- You can only delete user-created playlists; you cannot delete playlists on mediastores.

Use the following functions to create and delete playlists:

- *mme_playlist_set_statement()* — set the query statement to use when creating a playlist
- *mme_playlist_create()* — create a playlist
- *mme_playlist_delete()* — delete a playlist

## Deleting a playlist

To delete a playlist, call *mme_playlist_delete()*.

To following commands can be used with **`mmecli`** to use the user-created playlist functions:

- Create a playlist: **`mmecli playlist_create msid name`**
- Delete a playlist: **`mmecli playlist_delete plid`**

# *Chapter 8*

# Unsynchronized Media

The MME provides an extensive API for exploring and browsing unsynchronized mediastores.

## Exploring unsynchronized mediastores

The MME's mediastore explorer API can be used to provide the end-user with the following information from an unsynchronized mediastore, such as an iPod. It provides:

- the number of files and folders in the folder being explored
- metadata for the files in the folder, as requested

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

- Always request a number of items 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.

The explorer API includes the following functions, structures and constants:

- MME_EXPLORE_*
- *mme_explore_start()*
- *mme_explore_size_get()*
- *mme_explore_position_set()*
- *mme_explore_info_get()*
- *mme_explore_end()*
- *mme_explore_info_free()*
- **`mme_explore_hdl_t`**
- **`mme_explore_info_t`**
- *mme_explore_playlist_find_file()*

### Exploring a mediastore

To explore an unsynchronized mediastore:

1. Call *mme_explore_start()*, passing it the path to the folder you want to explore. This function returns a handle, **`mme_explore_hdl_t`**, which you can use with the other explorer functions to explore the folder.
2. Optional tasks:

- If you want to know the number of items of interest in the folder, call *mme_explore_size_get()*.
- If you want to start exploring the folder at a specified offset (other that the first item) call *mme_explore_position_set()* with the offset at which you want to start exploring.
- If you want to access metadata for the items you are exploring, call *mme_explore_position_set()* to set up the metadata types you want. For more information, see "Retrieving metadata from unsynchronized files" below.
- If you want to explore a playlist, call *mme_explore_playlist_find_file()* and check the values of the MME_EXPLORE_* flags.

**3** Begin exploring the items in the folder, starting at the specified offset, by calling *mme_explore_info_get()*. Each time you call this function, the offset will increment by one, so that the next call to *mme_explore_info_get()* retrieves information for the next item in the folder. When *mme_explore_info_get()* reaches the end of the folder, it returns NULL.

As you explore the folder, you can display to the user any metadata you have retrieved, and, if the user requests more complete metadata, call *mme_ms_metadata_get()*, *mme_metadata_extract_data()* and, if required, *mme_metadata_extract_string()* and *mme_metadata_extract_unsigned()* to retrieve and extract the metadata, then pass it up to the user.

Items retrieved by *mme_explore_info_get()* are presented as they occur; that is, they are *not* sorted or reorganized in any way. The items that are playable tracks, can be placed in file-based track sessions for playback. For information about how to create and modify file-based track sessions, see "Creating and modifying file-based track sessions" in the chapter Playing Audio.

**CAUTION:** This function may require considerable time to complete execution: with some mediastore types, it requires a *readdir()* of the entire item being explored.

### Retrieving metadata from unsynchronized files

The mediastore explorer API can be used to get the number of files and folders inside a specified folder on a mediastore, and, if requested, metadata. For optimal performance you should compose two different strings specifying the metadata to be retrieved:

- The string you pass to *mme_explore_position_set()* should set up *mme_explore_info_get()* to request only the metadata you will display to a user exploring the mediastore (for example, title and artist) — enough information to allow the user to decide if he or she wants more information about the track.
- The string you pass to *mme_metadata_extract()* should request more complete metadata, which the client application passes up to the user only in response to a specific request for more information.

It is possible to request complete metadata from *mme_explore_info_get()*, but doing so may prove slow, especially when communicating with external devices, such as iPods, that are connected via relatively slow ports.

There are two possible methods for composing the strings to retrieve metadata. Both methods use the values defined by the METADATA_* constants. You can compose your strings as comma-separated values according to either of the following models:

- **`char *types="title,artist,album"`**
- **`char *types=METADATA_TITLE","METADATA_ARTIST","METADATA_ALBUM`**

#### Reading and displaying explored file names

All filesystem APIs under QNX use UTF-8 character sets; and, with the exception of QNX4, all enforce its use, converting the character set on media to and from UTF-8 as required.

This characteristic of QNX filesystems means that when your client application is reading file and folder names from the explorer API it should assume that these names are in UTF-8 format. This rule includes filenames successfully converted from playlist file entries, but it does *not* include unconverted (raw) playlist file entries; the explorer API takes these entries directly from the playlist itself and does not convert them.

For information specific to displaying information from iPods, see "Displaying information from an iPod" in the chapter Working with iPods.

#### Filtering explored files

You can use the *mme_explore_position_set()* function's *flags* and *filter* arguments to filter the files examined and deliver only files of interest. Filtering is based on the values set in the *flags* argument, and can be done in two ways:

- *flags*=MME_EXPLORE_FILTER_INCLUDE — include only files with names that match the string referenced by the *filter* argument
- *flags*=MME_EXPLORE_FILTER_EXCLUDE — exclude all files with names that match the string referenced by the *filter* argument

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 );
```

For more detailed information, see *mme_explore_position_set()* in the *MME API Library Reference*.

## Using directed synchronization to browse mediastores

You can use the MME's directed synchronization capabilities to browse through mediastores. To let users browse through parts of a mediastore, call *mme_sync_directed()* with the path to the folder where you want to begin browsing. When the end user selects a new folder, call the function again with the new path.

For more information about directed synchronization, see "Directed synchronization" in the chapter Synchronizing Media.

# Chapter 9

# Metadata and Artwork

## In this chapter. . .

This chapter describes how to get metadata and artwork for your media files. It includes:

# Getting metadata

The MME provides a variety of methods for retrieving metadata. You can use the MME to retrieve metadata:

- for synchronized media, from the MME's **`library`** table, updated by the MME's second synchronization pass
- for unsynchronized media as well as synchronized media, directly from the mediastore
- from the **`nowplaying`** table
- from a remote source, such as a Gracenote server

## Getting metadata for synchronized media

You can provide your end users with information, such as album title, artist, and composer, for the currently playing file or track in a library-based track session by retrieving this metadata from the MME's **`library`** table.

To retrieve metadata for the currently playing file or track from the **`library`** table:

1. Call *mme_play_get_info()* to get the current file ID (*fid*).
2. Query the MME database for the metadata.

The sample query shown below retrieves title, album name, artist name, genre name, and composer name, assuming that you know only the *fid* for a track:

```
SELECT title, artist, album, genre, composer
    FROM library NATURAL JOIN (
    library_albums, library_artists, library_genres, library_composers)
    WHERE fid=%lld;
```

A library-based track session is a track session created with *synchronized* media files. For more information, see "About track sessions" in the chapter Playing Audio.

If an MP3 file contains more than one version of ID3 tags, the MP3 file parser parses the most recent tag and ignores the older tag version. That is, if a file contains both ID3v1 and ID3v2 metadata tags, the MP3 file parser ignores the ID3v1 tags. Only the metadata from the the ID3v2 tags is made available to the MME.

For information about getting metadata for tracks in a file-based track session; that is track sessions created with *un*synchronized media, see "Getting metadata for unsyncrhonized media" below.

## Getting metadata for unsynchronized media

The MME includes an API that can be used to retrieve metadata for files on an both *synchronized* and *unsynchronized* mediastores and devices.

This feature is particularly useful for quickly retrieving metadata for specific files from large mediastores or devices, such as iPods, that would take a long time to synchronize completely. For example, you can use this feature to retrieve metadata for a single file or a small number of files, to display in to a user exploring the contents of a mediastore or device.

This API includes the following functions, structures and constants:

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

The metadata extraction functions listed above require the filepath and filename of the file whose metadata is needed — for example, to display to a user who requests more information about a track on an iPod.

Thus, to retrieve and make available metadata for unsynchronized media, you must:

- Use the MME explorer API functions as described in the chapter Unsynchronized Media to get the filepath and filename of the file.
- Follow the steps described in "How to get the metadata" below to extract the metadata from the file.

### How to get the metadata

Once you have the path for the file whose metadata is needed:

1. Call *mme_ms_metadata_get()*, passing it the path to the file whose metadata you need, and the types of. It returns the **`mme_metadata_hdl_t`** with the metadata for the file.
2. Call *mme_metadata_extract_data()* to get the format of the metadata, and *mme_metadata_extract_string()* to extract metadata strings and *mme_metadata_extract_unsigned()* to extract unsigned metadata.
3. Use the metadata as required — for example, display to the end-user.
4. Call *mme_ms_metadata_done()* to complete the operation and release the metadata handle.

### Managing explorer structures and metadata handles

Metadata extraction from unsynchronized media requires that the client application manage the explorer information structures and the metadata and explorer handles it uses. The client application must:

- copy the **`mme_explore_info_t`** data structure
- manage the copies of this structure and the memory they require
- use *mme_metadata_alloc()* to copy metadata handles
- when it has finished with the metadata handles, deallocate the returned value from *mme_metadata_alloc()* by using *free()*

The client application must use *mme_metadata_alloc()* to copy the metadata handle. This *copied* metadata handle returned by *mme_metadata_alloc()* maintains valid information until the client application releases it.

Simply copying the **`mme_explore_info_t`** data structure does not guarantee valid information because:

- the pointer in the **`mme_explore_info_t`** data structure to the metadata handle may go out of scope before the client application has finished using the metadata object to which it refers
- **`mme_explore_info_t`** contains two pointers:
  - one pointer to the path member
  - one pointer to the metadata; this pointer may be null
- to ensure current and valid structures, the client application must have, as well as a copy of the **`mme_explore_info_t`** data structure, a copy of the metadata handle structure: **`mme_metadata_hdl_t`**; this handle is opaque and, therefore, can not be copied by the client application

The pointer to the path is a normal C-string; the client application:

- may use any method, such as *strdup()* or *strcpy()*, to copy the source string
- is responsible for managing the memory required for these copies

## Getting metadata from the `nowplaying` table

The **`nowplaying`** table may contain more complete metadata than is available in the **`library`** table — for example, when playing an iPod track session, because an iPod makes information, such as track duration, available only during playback.

When the MME starts playing a new track it updates the information in the **`nowplaying`** table and delivers an MME_EVENT_TRACKCHANGE event. When it updates metadata in the **`nowplaying`** table it delivers the event

MME_EVENT_NOWPLAYING_METADATA, which you can use to trigger queries to retrieve the updated metadata.
See also the description of the **`nowplaying`** table in the appendix: MME Database Schema Reference of the *MME API Library Reference*.

## Getting metadata from a remote source

The MME supports metadata from remote sources, such as CDText, Gracenote and MusicBrainz. To use these capabilities, you must change the MME configuration file, **`mme.conf`**, to enable the relevant MME modules and configure the specific behaviors required by your environment.

For information about how to configure the MME to support these modules, see "Metadata synchronizers" in the chapter Configuring Metadata Support in the *MME Configuration Guide*.

Support for these features may require special licensing. Contact QNX for more information.

## Metadata ratings

This MME supports metadata rating extractions for the following formats:

- MP3
- WMA

### MP3 files

If the ratings are available, the MME's MMF module extracts metadata ratings in MP3 files from the following tags:

| Tag | Frame |
|---|---|
| ID3v2.2 | POP |
| ID3v2.3 | POPM |
| ID3v2.4 | POPM |

### WMA files

If the ratings are available, the MME's MMF module extracts metadata ratings in WMA files from the **`Extended Content Description Object`**'s **`WM/SharedUserRating`** record.

### Ratings conversions

The MME stores metatdata ratings in the *rating* field in the:

- **`library`** table
- **`nowplaying`** table

The *rating* field in the **`nowplaying`** table makes a file's rating available even when the file does not have an entry in the **`library`**; for example, when the file is accessed through the MME's Explorer API.

Rating values are stored as follows:

- 0 — no metadata rating is available
- 1 to 255, with 1 for the lowest rating and 255 for the highest rating

The MME stores ratings from MP3 file ID3 tags without conversion. It converts WMA file **`WM/SharedUserRating`** record ratings from their 1 to 99 range to a 1 to 255 range.

The table below shows how the the MME's 1 to 255 rating system maps to WMA 1 to 99 rating system and to the WMA five star rating system:

| MME | WMA | Stars |
|---|---|---|
| 1-60 | 1-24 | * |
| 61-125 | 25-49 | ** |
| 126-190 | 50-74 | *** |
| 191-254 | 75-98 | **** |
| 255 | 99 | ***** |

# Getting artwork

The MME includes a "Load-on-Demand" API for retrieving metadata. This API is designed to support on-demand retrieval of all types of metadata, but it is currently implemented for artwork, such as album art images.

The Load-on-Demand metadata API can be used to retrieve artwork from synchronized or unsychronized media. It requires only that you specify the file for which you want to retrieve the artwork.

## Functions and data structures

The MME's Load-on-Demand metadata extraction API includes the following functions and data structures:

- *mme_metadata_create_session()* — create a new metadata session
- *mme_metadata_free_session()* — end a metadata session
- *mme_metadata_getinfo_current()* — retrieve metadata for the currently playing track
- *mme_metadata_getinfo_file()* — retrieve metadata for the specified file, based on its filepath
- *mme_metadata_getinfo_library()* — retrieve metadata for the specified file, based on its file ID in the **`library`** table
- *mme_metadata_image_load()* — load an image for a file
- *mme_metadata_image_unload()* — clear a specified image from temporary storage
- **`mme_metadata_image_url_t`** — the structure carrying the URL for an image
- **`mme_metadata_info_t`** — the structure that carries the metadata retrieved by any of the the *mme_metadata_getinfo_*()* functions
- **`mme_metadata_session_t`** — a metadata session identifier

Information for images stored in the MME's metadata image persistent cache is kept in the **`mdi_image_cache`** table.

For information about how to set configuration options for the MME's metadata extraction API, see the chapter *MME Configuration Guide* chapter Configuring Metadata Support.

## `libxml2.so` library and headers

MME releases include the **`libxml2.so`** library and appropriate headers; this library and the headers are required by clients of the MME in order to parse the metadata structures delivered by the MME metadata extraction functions.

The **`libxml2`** library delivered with the MME includes only a small subset of the full **`libxml2`** library; it includes only the modules required for reading, parsing and writing XML files. The **`xmlversion.h`** header file indicates exactly what functional modules are in the included library.

Documentation for **`libxml2`** is available at **`xmlsoft.org`**.

## Feature limitations

The current release support is limited, as follows:

- Support is limited to only the **`<image>`**/**`<format>`** metadata group
- The **`<image>`**/**`<format>`** metadata group does not produce the following image information:

  - MIME type
  - Description: “Front Cover”, “Back Cover”, etc.

- Extraction of external artwork is supported, but is limited to the following filenames:
  - **`album.jpg`**
  - **`ALBUM.JPG`**
  - **`folder.jpg`**
  - **`FOLDER.JPG`**
- Retrieval of embedded artwork from WMA files is not supported.

## Using the metadata extraction API

The MME’s metadata extraction API uses *metadata sessions*. A metadata session uses a metadata session identifier used by the metadata extraction functions.

Metadata extraction includes the following tasks:

1. Call *mme_metadata_create_session()* to create a metadata session and reserve the required system resouces.
2. Call one of the *mme_metadata_getinfo_*()* functions to retrieve the required metadata for a specified file and place it in the **`mme_metadata_info_t`** data structure.
3. If an image is available and required, call *mme_metadata_image_load()* to load the image into the image cache.
4. Call *mme_metadata_free_session()* to close the metadata session and free the system resources it was using.

The metadata extraction functions deliver metadata information as XML in the **`mme_metadata_info_t`** structure. For more information, and examples, see “XML content” on the **`mme_metadata_info_t`** structure reference page.

As required, the client application can also:

- use *mme_metadata_image_cache_clear()* to clear specified images from the metadata image cache
- use *mme_metadata_image_unload()* to clear a specified image from temporary storage

The MME supports extraction of album artwork from iPods for the currently playing track *only*. To retrieve album artwork from iPods, use the metadata extraction API as you do to retrieve artwork from other devices and mediastores.

See also “Retrieving artwork from iPods” in the chapter Working with iPods.

## Image pre-processing

The MME supports image pre-processing with the following capabilities:

- Decoding from the following source formats:
  - BMP
  - GIF
  - JPEG
  - PCX
  - SGI
  - TGA
- Encoding into the following target formats:
  - BMP
  - JPEG
- Image scaling and rotation

Images encoded into the BMP format may not be readable by some external applications.

### Enabling image pre-processing

The MME’s image processing capabilities are configured through the **`<MetadataInterface>`**/**`<ImageProcessing>`** element in the MME’s configuration file, **`mme.conf`**.

To enable image pre-processing, you must:

1. Configure the image processing library.
2. Enable one or more image processing modules.
3. Configure image processing profiles to be used by the MME’s metadata Load-on-Demand API.

**CAUTION:** If you do not enable the image processing module, or if there are no image processing modules configured, attempts to use a defined image processing profile through the MME's metadata interface will fail with an EINVAL error.

For information about how to configure and enable the MME for image pre-processing, see "Image pre-processing" in the chapter Configuring Metadata Support in the *MME Configuration Guide*.

### Using image pre-processing

The MME's metadata API implements the *mme_metadata_image_load()* function to load and, if requested, process images. Image processing is specified through the function's *image_format_profile* parameter. This parameter can be set to either -1 if no image processing is required, or the number of a pre-defined image processing profile. For more information about these profiles and how to configure then, see "Configuring image processing profiles" in the chapter Configuring Metadata Support in the *MME Configuration Guide*.

For more information about *mme_metadata_image_load()*, see the *MME API Library Reference*.

*Chapter 10*

# Playing and Managing Video and DVDs

## *In this chapter. . .*

This chapter describes how to work with track sessions and play audio media on the MME.

# Playing and managing video

The MME supports playback of MPEG4 files with H.264 video.

## Playing video files

Video files in the **`library`** table are identified by their *ftype* set to FTYPEVIDEO (2) or FTYPEAUDIOVIDEO (3). To play a video file, you need to:

1. Configure the MME for video support, by adding a video output device to the MME's **`outputdevices`** table, and setting values and behaviors for this device. For instructions, see "Configuring the MME for video support" in the chapter Control Contexts, Zones and Output Devices of this guide.
2. Create and set a track session that includes the video file you want to play, just as you would a track session with only audio files, or use the MME's Explore API to access the video file.
3. Start and manage playback, just like you would playback of an audio file: call *mme_play()*, *mme_stop()*, etc.

For information about:

- managing video playback, see "Managing video attributes" below.
- playing DVD-video, see "Playing and managing DVDs" below.
- configure **`io-media`** for optimal video performance, see "Configuring **`io-media`** for video " in the *MME Configuration Guide*.

## Managing video attributes

The MME provides an array of functions for managing videos while they are playing, including:

- *mme_video_get_status()*, which gets status information for video playback of any format. The MME indicates that there has been a change in video status by sending an MME_EVENT_VIDEO_STATUS event.
- *mme_video_set_angle()*, which sets the video angle for video playback. Before calling this function, use *mme_video_get_angle_info()* to get the current video angle.
- *mme_video_set_audio()*, which sets the audio stream for video playback in a control context, and *mme_video_get_audio_info()*, which gets information about audio settings for video playback.

- *mme_video_get_subtitle_info()* and *mme_video_set_subtitle()*, which get and set the subtitles for video playback.
- *mme_setlocale()*, which sets the preferred language for strings that indicate unknown media metadata, and *mme_getlocale()*, which gets the locale information.

# Playing and managing DVDs

The MME is designed to support the playing of:

- an entire DVD with navigation
- selected parts of a DVD, observing legal and other requirements and restrictions

DVD and video support is platform-specific. 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 an error with *errorno* set to ENOSYS.

Please contact us to discuss your DVD and video implementation requirements.

## DVD synchronization

When the MME synchronizes a DVD, it creates in the **`library`**:

- an entry for the entire DVD, with *ftype* set to FTYPE_DEVICE (5).
- entries for the various parts of the DVD.

The file IDs (*fid*s) for entire DVDs can be found by examining the **`library`** table entries where the *ftype* column has the value 5 (FTYPE_DEVICE from **`mme/interface.h`**).

The example below shows an SQL query to select all DVD entries in the library:

```
SELECT fid, title FROM library WHERE ftype=5;
```

## Playing DVDs

The MME supports playing an entire DVD or only a part of the DVD, subject to legal limitations that may either restrict access to some parts of a DVD or impose playback of other parts, such as the copyright notice and warning.

When playing a DVD-video, you can use *mme_video_get_info()* to get information, such as codec, capture format, and aspect ratio for the video, and *mme_video_set_properties()* to set the video properties for output.

### Playing an entire DVD

To play an entire audio or video DVD, simply include the *fid* for that DVD in your track session. With the entire DVD in the track session, you can call *mme_button()* to move around on the DVD, play media, and manage behavior. The MME delivers the appropriate events at all transitions during the DVD playback.

### Playing specific parts of a DVD

To play specific parts of an audio or video DVD (title, chapter, etc.), you can place the *fid*s for the parts of the DVD you want to play in a track session, then play the session. The MME will play only the specified part of the DVD, and deliver the event MME_EVENT_FINISHED when it has finished playing it.

Note that for DVD-videos legal restrictions may prevent playback of some parts of the DVD using this method. Similarly, attempts to use *mme_button()* to circumvent these restrictions will be rejected.

### Starting playback from a specific DVD title and chapter

You can start playback at a title and specific chapter of an audio or video DVD by calling *mme_seek_title_chapter()* when you would use a command button:

1. Create a track session with the file ID (*fid*) for the entire DVD.
2. Set the tracksession.
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.

Use *mme_get_title_chapter()* to get the number of titles and chapters in the current track, and the currently playing title and chapter numbers; use *mme_seek_title_chapter()* to seek to a specified title and chapter.

You can use these functions only if the MME_PLAYSUPPORT_NAVIGATION flag is set in the *support* member of the data structure **`mme_play_info_t`**. Call *mme_play_get_info()* to get this structure .

For sample code snippets, see the examples on the reference pages for *mme_get_title_chapter()* and *mme_seek_title_chapter()*.

## Setting the default preferred media language

To set the default preferred language for a media item, call *mme_media_set_def_lang()* with the *lang* argument pointing to a string with the requested language. You can also use *mme_media_get_def_lang()* to find out the currently set langauge.

See also Configuring Internationalization in the *MME Configuration Guide*.

## Managing DVD access

The MME API provides functions that facilitate managing access to DVDs, offering client applications the ability to get disk regions.

### Using DVD region codes

Region codes are used to set the regions for which a device is enabled, and to check the region of DVD-video discs before they are played. For example, if a user has a device enabled for regions 1 and 3, the HMI can check that a DVD-video disk is from one of these regions before allowing the user to play it.

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

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

It is the responsibility of the user application to set and track device regions, and to inform the end user through the HMI of conditions where the regions for a DVD-video disk are incompatible with the regions set for the device.

# Chapter 11

# Playback Errors

## In this chapter. . .

This chapter describes common playback errors and how to manage them.

The MME offers a number of methods for handling problems with media and with the environment in which it is used. These problems include damaged media, corrupt files, and vibrations in the environment.

Many of the options that configure the way the MME handles problems with media and its environment are configured in **io-media**. For more information, see **io-media** in the chapter MME Utilities Reference.

# CD drive timeout

When they encounter a read problem, many CD drivers (such as **devb-eide**) automatically retry the read until they time out. If the read problem is due to vibrations, there is a good chance that the vibrations will cease before the time out and that the playback will continue successfully. However, if the read problem is due to scratched or otherwise damaged media, the read will continue to fail until the drive times out and delivers an EIO error.

This behavior indicates that driver read timeouts should be configured differently depending on the environment in which a drive is installed:

- In an environment (such as a stationary installation) with little chance of vibration errors, read errors will almost always be caused by defective media. There is, therefore, no need for a long retry period, and the read timeout period should be relatively short.
- In moving environments (such as in automobiles, airplanes, or trains), read errors will often be caused by vibrations and not by problems with the media storage device (the CD or DVD). A relatively long retry period will allow the drive to recover and continue playing.

To set the timeout period for a drive, use the command-line options for your CD drive. For more information about setting the driver time-out period, refer to the documentation for the CD driver or drivers you are using.

# Playback buffering

To permit uninterrupted playback to the user in the event of recoverable read errors (such as errors caused by vibrations) **io-media** buffers data. By default, **io-media** queues 49 buffers of data for playback. The data is buffered before decompression, so the play time for these buffers varies according to the amount of compression used for the media tracks being played. Total buffered play time available with 49 buffers is, approximately:

- CDDA — 10 seconds of buffered play time.
- MP3 — 100 seconds of buffered play time.

Other media formats have comparable buffered play times, depending on the level of data compression.

You can specify the number of buffers available for queued playback via the **io-media** configuration file.

# Playback read error recovery

If **io-media** encounters a read error from a CD drive and it cannot recover within the time set in the device driver timeout configuration, the MME will attempt to skip ahead to a different part of the track and continue playing. If the MME encounters a read error at the new location, it increases the skip time and attempts to read the media at a third location. If this read fails, the MME repeats the process until one of the following conditions occurs:

- While skipping forward, the MME reaches the end of the track. In this case the MME reports a normal end of track.
- Attempts to read the track continue failing until the number of skip forwards exceeds the maximum allowed. In this case, the MME reports a fatal read error.

Note that **io-media** may skip ahead due to damaged (scratched) media or to vibrations in the environment; **io-media** knows only that the CD drive reported a read error and that it needs to skip to another part of the track it is trying to play.

You should configure **io-media** at startup to define how it behaves in the event of playback read errors. Configurable settings are:

- enable skip ahead — enable or disable skip ahead on playback read error
- skip seek time — the number of milliseconds to skip forward in a track when attempting to recover from a read error.
- increment percent — the percent of the previous skip seek time to seek forward repeatedly until playback is possible
- maximum retries — the maximum number of times to skip before failing.

For example, with the skip seek time set to 120 milliseconds and the increment percent set to 50, if **io-media** is unable to play from a point on a track, it will:

1. Skip forward 120 milliseconds and attempt to resume playback.
2. If playback fails at the new position, **io-media** will add 50 percent to the skip time (60 milliseconds), and skip forward a second time (180 milliseconds forward from the original point of failure).
3. If playback fails a third time, **io-media** will add 50 percent to the last skip time (90 milliseconds, for a total skip of 270 milliseconds from the original point of failure).
4. If the failure persists, **io-media** will continue to skip forward in 50 percent increments until it is able to successfully play the track or it reaches the end of the track.

When **io-media** is able to resume playback, it resets the skip seek time to the configured time.

If a track is being played backwards, and **io-media** is configured to skip on error, **io-media**:

- skips backwards
- stops and reports a read failure at the beginning of the track

For more information about configuring playback read error recovery behavior, see "Configuring how the MME handles playback read errors" in the *MME Configuration Guide*. For information about how to configure **io-media**, see **io-media** in the *MME Utilities Reference*.

# Stopping playback after repeated playback failures

When the MME receives the instruction to play a track (*mme_play()*, *mme_next()*, or *mme_prev()*), it attempts to start playback of the requested track. If the requested track is not playable because the media is damaged (e.g. a scratched CD), the MME attempts to play the next track, continuing until it finds a playable track or it has tried and failed to play every track in the track session. This behavior prevents the MME from processing any other request until it has found a playable file or attempted to play every track in the track session.

To stop the MME from attempting to play every track in a track session, you can call *mme_stop()* to stop playback, and switch playback to another track session on another mediastore, such as the HDD, then eject the bad mediastore. You can determine the conditions under which your client application will use *mme_stop()* to stop a track session. For example, your client application can stop a track session after it receives several MME_PLAY_ERROR_READ and/or MME_PLAY_ERROR_CORRUPT events in sequence.

# Marking unplayable files

The MME provides a flag that identifies unplayable files. Your client application can use this flag to filter unplayable files from track sessions. If the MME is unable to play a track, it:

- may mark the track as unplayable by setting the track's *playable* field in the **library** table to 0 (zero)
- delivers an MME_PLAY_ERROR* event

### What files are marked as “unplayable”

The *playable* field does not apply to files that the MME can start to play but on which it encounters errors later during playback. The MME marks only files for which it cannot initiate playback because, for example, the file is invalid, the codec for the file format is not available, or DRM forbids playback of the file.

For errors that occur after playback starts (e.g. a track is so badly scratched in the middle that **`io-media`** gives up trying to read it, or an MP3 file is corrupt somewhere in the middle), the MME doesn’t send an MME_PLAY_ERROR* event and the playable field isn’t set to 0 (zero). In these cases, the MME delivers an MME_EVENT_TRACKCHANGE event when it goes to play the next track.

### Skipping “unplayable” files

To ensure that your client application does not attempt to play files marked as unplayable, you can include the clause **`WHERE playable=1`**, in the SQL **`SELECT`** statement you use to build your track sessions.

The **`<SkipUnplayable>`** configuration element can be used to have the MME automatically skip unplayable files without sending any error messages to the client application. For more information, see “Automatically skip files marked as unplayable” in the *MME Configuration Guide*.

## Handling damaged media

To support handling of damaged media, when it encounters read errors, the MME returns the following events, depending on the type of read error condition encountered:

- MME_EVENT_PLAY_WARNING — **`io-media`** encountered a read error and is attempting to skip forward past the bad section of the track.
- MME_EVENT_PLAY_ERROR — **`io-media`** has surpassed the maximum skip forward on read error attempts, and has given up attempting to read the track.
- MME_EVENT_TRACKCHANGE — while attempting to skip forward past a bad section of a track, **`io-media`** has advanced beyond the end of the track. The MME has performed a track change and is attempting to play the new track.
- MME_EVENT_FINISHED — while attempting to skip forward past a bad section of a track, **`io-media`** has advanced beyond the end of the track, and there are no more tracks to play in the track session.

To support this behavior, the enumerated type **`mm_warnings_t`** has the following values:

- MM_WARNING_READ_TIMEOUT — the source was slow and a read timed out.
- MM_WARNING_READ_ERROR — there was a read error, and the operation is trying to recover.

### Synopsis

```
typedef enum e_mm_warnings {
MM_WARNING_READ_TIMEOUT = 0x1,
MM_WARNING_READ_ERROR   = 0x2,
} mm_warnings_t;
```

# Chapter 12

# Copying and Ripping Media

## In this chapter. . .

The MME provides capabilities for copying and ripping media. Ripping is the process of reading files from a mediastore, changing the format of these files into another format if required, then writing the files in their new format to a mediastore or other storage device. Copying media is simply ripping media and writing the destination files in the same format as the source files.

# About media copying and ripping

You can instruct the MME to rip media from one or several mediastores to any writable mediastore.

## The copying and ripping process

When the MME performs a ripping operation, it looks through its copy queue, stored in the **copyqueue** table, for the IDs of the files to copy or rip, as well as other information it needs for the operation.

If it is copying media, the MME will copy files along with all their metadata. Depending on how it is configured, the MME will either copy all media to one folder or preserve the original folder paths for the copied files on the destination mediastore. For more information, see "Managing folder paths" below.

The MME will copy or rip a file only if it isn't being played or synchronized. Similarly, the MME will abort a copy or ripping operation if during the operation the source file is requested for playback.

If it is ripping media, the MME uses a metadata database, such as Gracenote, AMG or CD-Text, to add to the metadata of the new ripped file, and uses the ripping template defined through *mme_mediacopier_add()* to retrieve the folder paths to use when writing the ripped files. For more information, see "Copy folder paths and ripping templates" below.

When it has received notification that a copy or ripping operation has completed, your client application should use *mme_mediacopier_clear()* to clear the copy queue so that subsequent copying or ripping operations don't copy or rip the same files twice.

## Monitoring progress and playback

Your client application should use *mme_mediacopier_get_status()* to retrieve the status of media copying and ripping. Copying provides the number of bytes to be copied and the number of bytes copied. Ripping provides the total play time of the media track and the amount of play time ripped. Always check for MME media copying and ripping events (MME_EVENT_MEDIACOPIER_*) to check on the progress of copying and ripping operations, and to know when they are completed.

## Priority background ripping

If priority background ripping is set, the MME plays back copied or ripped files from the copied or ripped files, not from the source file.

#### Event delivery during priority background ripping

During priority background ripping operations, delivery of:

- MME_MEDIACOPIER_COMPLETE indicates that there is nothing left in the copyqueue, and that the mediacopier is stopping.
- MME_MEDIACOPIER_DISABLED indicates that the mediacopier has stopped because:
  - it received a stop request from the *mme_mediacopier_disable()*
    or
  - it is unable to rip any of the tracks currently in the copyqueue

# Copying and ripping media

Copying or ripping with the MME requires that you perform the following tasks, in order:

1. Check and, if necessary, set the media copying or ripping mode.
2. For copy operations, decide if you want to preserve the original folder path or use a new path on the target mediastore; for ripping operations, get the ripping template.
3. Prepare the copy queue with the files you want to copy or rip.
4. Start the operation.

When it has finished copying or ripping all files in the copy queue, the MME delivers the event MME_EVENT_MEDIACOPIER_COMPLETE.

## Setting the copy or ripping mode

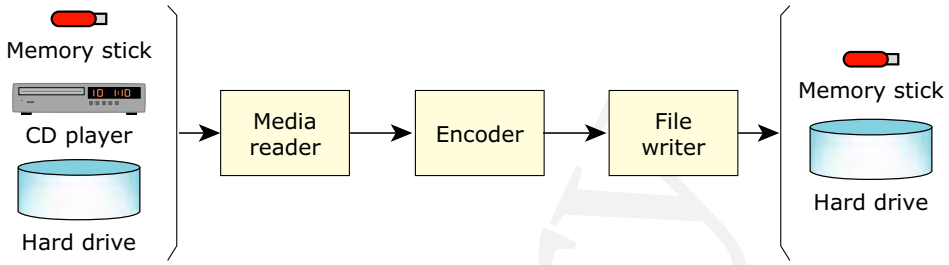
To check the copy and ripping mode, use *mme_mediacopier_get_mode()*; to set the mode, use *mme_mediacopier_set_mode()*.

The MME offers two modes for media copying and ripping:

- background
- priority background

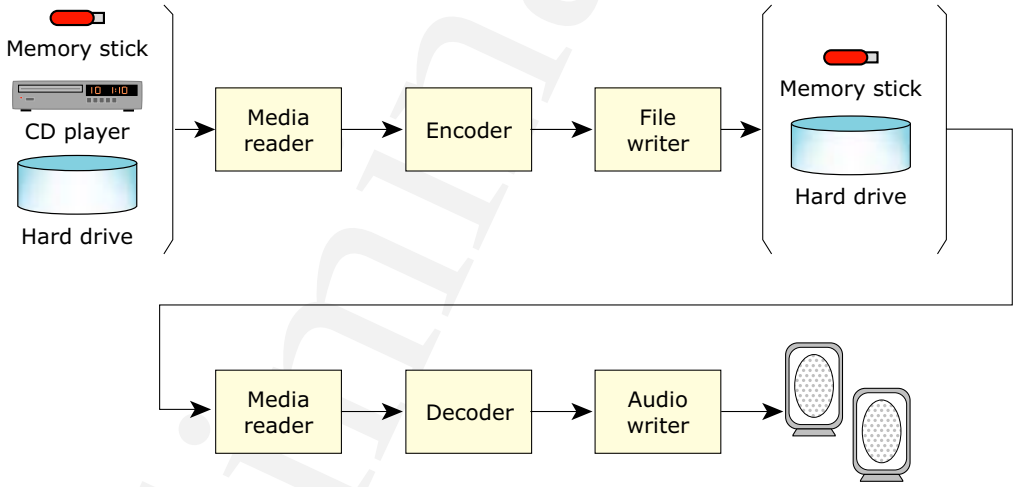
Both the background and the priority background modes are non-blocking. That is, after you set up the copy or ripping operation and call *mme_mediacopier_enable()* to start it, the MME starts copying or ripping in the background and your application can go on to perform other tasks. Use *mme_mediacopier_get_mode()* to retrieve the current media copying and ripping mode. Note that to use priority background media copying and ripping, you must enable this mode in the configuration file **`mme.conf`** before starting the MME. For more information, see the chapter Configuring Media Copying and Ripping in the *MME Configuration Guide*.

The figure below illustrate background ripping:




*MME background ripping operation.*

The figure below illustrates priority background ripping with playback:




*MME priority background ripping operation with playback.*

## Copy folder paths and ripping templates

If you are copying media, you can elect to preserve folder paths; if you are ripping media, you can use ripping templates to define how ripped files will be organized on the target mediastore.

### Managing folder paths

The MME can be configured to maintain folder paths by default when copying files by setting the **`<PreservePath>`** element in the **`mme.conf`** file: **`<PreservePath enabled="true"/>`**. For more information, see the chapter Configuring Media Copying and Ripping in the *MME Configuration Guide*.

If it is configured or instructed to preserve folder paths, when it performs a file copy, the MME:

- recreates in the destination the folder paths for the copied files

- updates the **`folders`** table with entries for the newly created folders

For example, if a source file is located in **`/fs/usb0/mymusic/albums/pinkfloyd/wall.mp3`**, the MME will copy it to **`/media/drive/ripped/mymusic/albums/pinkfloyd/wall.mp3`**.

The MME also supports the dynamic setting of folder paths during a copy, which you can select later in the process, when you call *mme_mediacopier_enable()*:

- To ignore the original path and create a new path during copy, set the *flags* argument in *mme_mediacopier_add()* to MME_MEDIACOPIER_NONE.
- To preserve the original folder path for the copied media folders and files, set the *flags* argument in *mme_mediacopier_enable()* to MME_MEDIACOPIER_PRESERVE_PATH.

Folder path functionality applies only to media copy operations, *not* ripping operations.

#### Using the $*PRESERVE_PATH* template strings

The $*PRESERVE_PATH templates strings may only be used for copying operations. They are *not* supported for ripping operations.

These template strings:

- override the global preserve path configuration set with the **`<PreservePath>`** element in the MME configuration file
- must be the last templates strings in a path
- may optionally be terminated with a "/" character
- are for use on a per copy basis; they must not be used in the global default folder name

If you use $PRESERVE_PATH or $NO_PRESERVE_PATH with no additional path information provided, the operation uses the the global default copy destination folder:

- **`/$NO_PRESERVE_PATH`** copies to the root of the target mediastore
- **`/$PRESERVE_PATH`** builds the path from the source mediastore on to the destination mediastore

$PRESERVE_PATH_AFTER is used in the destination folder name to modify the source path when it is appended to the destination folder. When it builds the destination path, the MME copy operation searchs for and discards from the source path the characters after $PRESERVE_PATH_AFTER: (note the colon ":"). It discards characters from the beginning of the source path up to and including the characters after $PRESERVE_PATH_AFTER:.

For example, to transform the source path **`/fs/pfs0/Music/Artist/Album/Song.mp3`** into **`/copy_dir/Artist/Album/Song.mp3`** on the copied mediastore, you can use either of the following destination paths:

- **`/copy_dir/$PRESERVE_PATH_AFTER:/fs/pfs0/Music/`** — explictly define all characters to be removed from the destination path
- **`/copy_dir/$PRESERVE_PATH_AFTER:/Music/`** — discard all characters up to an including the stated string

### Setting and using ripping templates

The function *mme_mediacopier_add()* uses the data structure **`mme_mediacopier_info_t`** to set templates that define how the MME names ripped files and where it places them in folder structures. If a ripping template is set, when the MME rips media, it automatically names the ripped files and places them in the locations defined by the template, building the folder structures and filling in appropriate information based on the metadata for the ripped files.

For example, the following strings defined for the templates:

| | |
|---|---|
| Folder | **`/$ARTIST/$ALBUM/`** |
| File name | **`/$title/`** |

would yield (based on the metadata) the following folders and file names:

- **`/Katia Guerreiro/Tudo ou Nada/Despedida.mp3`**
- **`/Pearl Jam/Ten/Oceans.mp3`**
- **`/Pearl Jam/Ten/Alive.mp3`**
- **`/U2/Joshua Tree/Exit.mp3`**

Use *mme_mediacopier_add()* to set ripping templates at any time before a ripping operation and to get the template you want to use before a ripping operation. For example, you could define one template to organize media by artist, album and title, and another to organize media by genre, year and artist, and offer the end-user the option of ripping media using either template. Or, your HMI could let the end-user build additional templates and store them.

## Building the copy queue

After you have set the copy or ripping mode you want to use, and how you want to handle folder paths for media copies or the organization of ripped files, you need to compose an SQL query statement and call *mme_mediacopier_add()* with this statement to:

- build up the list in files to be copied or ripped in the copy queue

- tell the MME whether this is a media copy or a ripping operation
- set the destination mediastore for the operation
- for ripping operations, set the target format for the media.

The example below shows how to add files to the copy queue, ensuring that there are no duplicate entries. The SQL query could be something such as **"SELECT fid FROM library WHERE msid=***msid_num***"**, where *msid_num* is the mediastore ID of the CD you want to copy, and you want to select all tracks from the CD for ripping.

```
// If none of this CD's tracks were ripped before, we make sure
// that we add them to the copyqueue.
if( init_copyqueue ) {
    mme_mediacopier_info_t info;

    // Clear the copyqueue.
    // We do this to prevent copying tracks multiple times.
    mme_mediacopier_clear( mme );

    // Set up rip destination and encoding
    info.dstfilename    = NULL;
    info.dstfolder      = NULL;  // use the defaults
    info.dstmsid        = 0;
    info.encodeformatid = 2;     // 2 is 'wav',
    // see the 'encodeformats' table.

    if( -1 == mme_mediacopier_add( mme, &info, sql, 0 ) ) {
        perror( "mme_mediacopier_add()" );
        return;
    }
}
```

Note how in the example above, the client application calls *mme_mediacopier_clear()* to clear the copy queue before adding files to it. For more information, see "Managing the copy queue" below.

### Updating metadata

The MME can be configured to synchronize files with inaccurate metadata before copying or ripping them. If this option is set, when the MME prepares to copy or rip a file, it:

- Checks the *accurate* field in the **library** table for the file.
- If this field is set to 0 (the metadata is not accurate), before copying or ripping the file, the MME synchronizes the source file so its metadata is accurate, ensuring the accuracy of the metadata in the copied or ripped file.

To configure the MME to synchronize files with inaccurate metadata before copying or ripping them, make sure that the **<Copying>** element **<UpdateMetadata>** in the MME configuration file (**mme.conf**) is set to **true**. This is the default setting.

#### Completing unknown metadata

To add specified metadata strings when metadata is not known, build the copy queue with the function *mme_mediacopier_add_with_metadata()* instead of *mme_mediacopier_add()*.

## Starting media copying or ripping

To start a media copying or ripping operation, call *mme_mediacopier_enable()* instructing it to either copy or rip the media listed in the copy queue. This function will read through the copy queue in the **copyqueue** table and either copy or rip the files, as directed.

## Stopping media copying or ripping

To stop a media copying or ripping operation, call *mme_mediacopier_disable()*.

## Behavior when media copying or ripping encounters an error

If the MME is unable to copy or rip a file, it:

- delivers the event MME_EVENT_MEDIACOPIER_SKIPFID or MME_EVENT_COPY_ERROR
- moves to the next entry in the **copyqueue** table

You may want to remove entries for skipped files from the **copyqueue** so that the MME does not attempt to copy or rip them the next time you begin a media copying or ripping operation.

You can use the **<DeleteOnNonRecoverableError>** element in the MME configuration file to have the MME automatically delete from the copy queue entries for files that cause unrecoverable errors.

## Behavior when a mediastore is removed

If the mediastore from which files are being copied or ripped is ejected during the operation, the MME will deliver the event MME_EVENT_COPY_ERROR (MME_COPY_ERROR_NOTSPECIFIED) and an event MME_EVENT_MEDIACOPIER_SKIPFID event for the track being ripped, as well as the next few tracks in the copy queue until the MME detects that the mediastore was ejected.

When the MME detects that the mediastore was removed from the system, it delivers the event MME_EVENT_COPY_ERROR (MME_COPY_ERROR_DEVICEREMOVED), and removes the partially ripped file from the destination mediastore.

# Managing the copy queue

Stopping a media copying or ripping operation does not affect the **copyqueue** table. You should ensure that your client application manages the copy queue so that you do

not inadvertently copy or rip files left in the queue by an earlier media copying or ripping operation. At the start of every media copying or ripping operation, you should call *mme_mediacopier_clear()* to remove all entries in the **copyqueue** table.

You can also remove specific items from the copy queue by calling *mme_mediacopier_remove()*. This feature allows you to offer the end users functionality such as the ability to review a list of media to be copied or ripped and add or remove individual entries as desired *before* starting a media copying or ripping operation.

# Modifying media metadata

The MME provides *mme_metadata_set()* so your client application can provide the end-user with the ability to modify the metadata for copied and ripped media. You can design an HMI interface that displays media metadata and accepts input of corrections or additions from the user, then pass the input to the *mme_metadata_set()*, which will write the modified metadata to both the media file (where applicable) and the MME database.

# Chapter 13

# External Devices, CD Changers and Streamed Media

## In this chapter. . .

The MME supports playing streamed media, as well as media on mediastores on external changers. This chapter describes:

- Getting and setting device options
- Working with external CD changers
- Working with internet streamed media
- Audio input playback

# Getting and setting device options

The MME supports getting and setting device option configurations, even if the MME does not know about the options:

- Device option configuration API
- Getting and setting device configuration values
- Determining the iPod connection and capabilities

## Device option configuration API

The API or getting and setting device option configurations uses the following functions, data structures, enumerated types and events:

- *mme_device_get_config()*
- *mme_device_set_config()*
- MME_EVENT_MEDIA_STATUS
- **mm_media_status_t**
- **mm_media_status_event_t**
- **mm_media_status_reason_t**

## Getting and setting device configuration values

The *mme_device_get_config()* and *mme_device_set_config()* functions get and set configuration option values for devices accessed through MediaFS. The MME does not need to know about the options or their settings.

At present, you can use *mme_device_get_config()* to get configuration values for iPod devices and Bluetooth devices that use a Temic stack, and *mme_device_set_config()* to set options on iPod devices, with these constraints:

- *mme_device_get_config()* returns all configuration options for a device; individual elements or attributes can *not* be specified
- *mme_device_set_config()* sets only one XML element attribute at a time; to set multiple attributes, you must call the function once for each attribute

### Supported interfaces

At present, the MME supports getting and setting interface options for two types of interfaces:

- USB devices plugged into the system
- devices accessed through a device driver (such as, for example, a QNX resource manager) running on the system

The **<interface>** configuration element for USB devices uses the following attribute template:

```
<interface type="usb" path="USB_bus_number" devno="USB_device_number" \
vendorid="USB_vendor_id_number" productid="USB_product_id_number"/>
```

The **<interface>** configuration element for devices accessed through a device driver uses the following attribute template:

```
<interface type="device" path="/fsys/path/to/device/resmgr"/>
```

### Getting configuration values from iPods

To get the configured settings for an iPod:

1. Reserve a buffer for the information that will be returned from the device.
2. Call *mme_device_get_config()*.

For example, for an iPod with the mediastore ID 2:

```
char buf[1000];

mme_device_get_config(hdl, 2, "/", 0, sizeof(buf), buf);
```

The function will fill the buffer with the device information, which will be presented in a format like the following for an iPod using USB transport:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<device api_version="1">
 <interface type="usb" path="1" devno="1" vendorid="0x5ac" productid="0x5ac"/>
 <media>
  <iPod>
   <capabilities>
    <video/>
    <digitalaudio/>
   </capabilities>
   <transport value="usb_ipod" valuetype="text"/>
   <preferences>
    <video value="ask" valuetype="enum" modifiable="no" alternatives="off,on,ask"/>
    <screen value="fit" valuetype="enum" modifiable="yes" alternatives="fill,fit"/>
    <format value="ntsc" valuetype="enum" modifiable="no" alternatives="ntsc,pal"/>
    <connection value="composite" valuetype="enum" modifiable="no" alternatives="none,composite,svideo,component"/>
    <caption value="off" valuetype="enum" modifiable="no" alternatives="off,on"/>
    <ratio value="full" valuetype="enum" modifiable="no" alternatives="full,wide"/>
    <subtitle value="off" valuetype="enum" modifiable="no" alternatives="off,on"/>
    <audioalt value="off" valuetype="enum" modifiable="no" alternatives="off,on"/>
   </preferences>
  </iPod>
 </media>
</device>
```

Or the following for an iPod using serial transport:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<device api_version="1">>
<interface type="device" path="/net/groytest2.ott.qnx.com/dev/ser1"/>
 <media>
  <iPod>
   <capabilities>
    <video/>
    <digitalaudio/>
   </capabilities>
   <transport value="ser_ipod" valuetype="text"/>
   <preferences/>
  </iPod>
 </media>
</device>
```

### Getting configuration values from Bluetooth devices

The *mme_device_get_config()* can be used to retrieve device configuration settings from Bluetooth (A2DP) devices.

To retrieve this information from a Bluetooth device, simply call *mme_device_get_config()* as you would for an iPod device, but with the mediastore ID set for the Bluetooth device. The information in buffer filled in by the call will look something like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<device api_version="1">
  <interface type="device" path="/dev/wms/player1"/>
  <media>
    <AVRCP>
      <version>
        <major value="1" valuetype="num"/>
        <minor value="3" valuetype="num"/>
      </version>
    </AVRCP>
  </media>
</device>
```

### Setting configuration values on an iPod

The *mme_device_set_config()* allows you to change the iPod preferences that were configured when the iPod driver was started. It sets configuration values by setting the attributes for sub-elements inside the **`<preferences>`** attribute.

Each element inside the **`<preferences>`** has the following attributes:

- *value* — the current setting
- *valuetype* — the type of setting (currently all settings are enumerated values; that is, a choice from a fixed list)

- *modifiable* — determines if the other attributes can be changed
- *alternatives* — possible values for the settings

At present, the only element with modifiable attributes is **`<screen>`**.

To modify an iPod's screen zoom mode:

1. Call *mme_device_set_config()* with the *xpath* argument set to the path to the element's *value* attribute, and the *newvalue* set to the desire value.

For example:

```
mme_device_set_config(hdl, 2, "/device/media/iPod/capabilities/screen@value", "fill", 0);
```

On an iPod:

- "fill" means stretch the image *without* altering the aspect ratio. That is, do not haven; that is, the image will have no vertical or horizontal black bars (no letterbox or pillar box), but it may be cropped to make it fit the iPod screen
- "fit" means that the image is *not* cropped, but the image may be framed by horizontal or vertical black bars

## Determining the iPod connection and capabilities

To determine how an iPod is connected, simply call *mme_device_get_config()*, and in the returned buffer with the device information, check the following elements and attributes:

- Check the **`<transport>`** element's *value* attribute:
  - **`"ser_ipod"`** — the iPod is using serial transport
  - **`"usb_ipod"`** — the iPod is using USB transport
- Check the **`<capabilities>`** sub-elements. For example, if **`<digitalaudio>`** is present, then the iPod supports digital audio.

# Working with external CD changers

The MME includes several features that facilitate working with external CD changers:

- The constants STORAGETYPE_MEDIAFS_* support different mediastore types on the same device.
- The enumerated types **`mme_mode_random`** and **`mme_mode_repeat`** include values to support random and repeat modes for folders and subfolders.

- The MME supports track changes initiated by external CD changers. When the MME receives notification that an external CD changer has changed tracks, it:
  - identifies the new currently playing track
  - updates the **`nowplaying`** table with the file ID (*fid*) and metadata for the currently playing track

A random or repeat mode setting works only if the external device supports the setting. If the external device does not support the requested setting, the MME logs a warning and continues playback.

For more information about building systems that use external CD changers, contact your QNX representative.

# Working with internet streamed media

The MME supports playback of internet streamed media, including:

- HTTP streamed audio, such as SHOUTcast and Icecast. Currently supported formats are:
  - AAC
  - MP3
- RTP streamed video from an IP camera. Currently supported formats are:
  - MPEG4-ES (MPEG4 video elementary stream)

## RTP streamed media

The MME can output RTP streamed media from a camera. Support is presently limited to ouput of the media stream; features such as pan and tilt control are not supported.

To access the RTP stream, pass the RTSP (Real Time Stream Protocol) access URL of the camera to **`io-media`**.

For example, an Axis 207 network camera on a network could be accessed with the following URL: **`rtsp://10.42.108.95:554/mpeg4/1/media.amp`**, where:

- **`10.42.108.95`** is the IP address of the camera on the network
- **`554`** is the IP camera access port number

## Configuring the MME to support streamed media

To configure the MME to support streamed media, you must:

1. Add a rule to the MCD to detect internet connections.
2. Enable an internet slot in the **`slots`** table.

### Add a rule to the MCD to detect internet connections

To add a rule to the MCD to detect internet connections, simply add a rule to the MME's MCD configuration file to have the MCD look for internet connections. For example:

```
[/dev/socket]
Callout     = PATH_MEDIA_PROCMGR
Argument    = /proc/mount
Priority    = 11,10
Start Rule  = INSERTED
Stop Rule   = EJECTED
```

For more information about MCD rules, see "Configuring the **mcd** utility" in the *MME Configuration Guide* chapter Configuring Device Support.

### Enable an internet slot in the `slots` table

After you have configured the MCD to look for internet connections you must configure the **slots** table to support internet connections. For example:

```
INSERT INTO slots(path,zoneid, name, slottype)
    VALUES('/dev/socket', 1, 'INTERNET', 10);
```

For more information about configuring the **slots** table, see "Configuring the **slots** table for supported devices" in the *MME Configuration Guide* chapter Configuring Device Support.

## Playing streamed media

If you have configured the MME to support internet connections, its **mediastores** table should have an entry for an "internet" mediastore. You can check this by querying the database from the commandline. For example:

```
qdbc -d mme "select msid,slotid,name,mountpath from mediastores"
```

One of the returned lines should return values something like:

```
|2|13|Internet|/dev/socket|
```

Media streams cannot be synchronized, so to play it you should played in a *file-based* track session:

1. Create a file-based track session by calling *mme_newtrksession()* with the *mode* argument set to MME_PLAYMODE_FILE.
2. Set the track sesssion by calling *mme_settrksession()*.
3. Append the the HTTP stream by calling *mme_trksession_append_files()* with the *filename* argument referencing to the HTTP stream.

   or:

4. Append the RTP stream by calling *mme_trksession_append_files()* with the *filename* argument set to the URL of the RTP stream from the Axis camera server: **"rtsp://10.42.108.95:554/mpeg4/1/media.amp"**.

5 Proceed with playback.

The **SELECT** statement used to create the track session query for media where *ftype*=5 (media that can be played as one file).

For example, with the **mmecli** commandline utility, you might play streamed media as follows:

```
# mmecli newtrksession f "select fid from library where ftype=5 and msid=2"

# mmecli settrksession 1

# mmecli trksessionview_append_file 1 2 http://www.playsong.com/song

# mmecli play
```

# Audio input playback

The MME now supports playback of audio inputs on a system by treating audio inputs as type of mediastore.

This feature uses a new mediastore type and a new slot type:

- MME_STORAGETYPE_SND_INPUT
- MME_SLOTTYPE_SND_INPUT

## Configuring the MME to recognize audio input “mediastores”

To have the MME recognize audio input “mediastores”, you must:

- add an appropriate entry to the **slots** table
- configure the MCD for audio input “mediastores”

### Configuring the slots table

To use audio input “mediastores”, you must add an entry to the **slots** table that sets:

- the path to the location of the audio input
- the slot type to 11

For example:

```
INSERT INTO slots(path, zoneid, name, slottype) VALUES('/dev/snd', 1, 'sn
```

### Configuring the MCD

You must also configure the MCD to tell the MME about the appearance of audio input “mediastores” by adding a section like the following to the MCD configuration:

```
[/dev/snd]
Callout = PATH_MEDIA_SCAN
Argument = 5000
Priority = 11,10
Start Rule = INSERTED
Stop Rule = EJECTED
```

Since **/dev/snd** is normally permanent, the argument and priority values may be changed. The section name must be the path to the audio input devices.

An audio input “mediastore” is not synchronizable; an explicit attempt to synchronize it will result in a MME_EVENT_SYNCABORTED error.

## Playing media from an audio input “mediastore”

Media from an audio input “mediastore” is played in a file-based track session. Once playback has begun, however, it will continue until explicitly stopped because it will never reach an end of file.

To play media from an audio input “mediastore” you must know the exact path to the mixer output you want to play. Generally tis path is something like **pcmC0D0c#Mic In**.

You may learn what is available by:

- doing an **ls** of the path to the location of the audio input
- looking at the output of **mix_ctl**

Once you know the path of your audio input, simply create a file-based track session and play the media

1. create a file-based track session, using the **snd** “mediastore”’s device file ID.
2. Set the track session for playback.
3. Call *mme_trksession_set_files()* to set the input to the track session.
4. Play the track session.

# Chapter 14

# Working with iPods

## In this chapter. . .

This chapter describes:

- Installing MME components for external media players
- Connecting to and using iPods
- Link kit for iPod authentication

For information about how to get configuration values from an iPod device, see "Getting and setting external device options" in the chapter External Devices, CD Changers and Streamed Media.

# Installing MME components for external media players

If you want use an external media player, such as an iPod or a PlaysForSure-enabled device, you need to:

1. Install the runtime files that support these devices. These installations may require special licenses.
2. Use **`iofs-ipod`** or **`iofs-pfs`**, depending on the type of media player.

For more detailed instructions, see the QNX® Aviage Multimedia Suite *Installation Note*.

# Connecting to and using iPods

This section describes how to connect to iPod devices, and how to use the MME to interface to and manage iPods, where the behavior of iPods requires an approach different from that used for other devices.

- Required components
- Authenticating iPods
- Connecting to iPods
- Detecting iPods
- Removing iPods
- Synchronizing iPods
- Playing media on iPods
- Displaying information from an iPod
- Uploading splash screens to iPods
- HD radio tagging

## Required components

The table below lists the non-standard components you need to connect to and use iPod devices from a QNX system:

| Component | Serial | USB | Description |
|---|---|---|---|
| Licenses | Yes | Yes | See "Licenses" below. |
| Authentication chip | Recommended | Yes | See "Authenticating iPods" below. |
| **`deva-ctrl-ipod.so`** | Depends on configuration | Yes | The iPod audio driver. |
| **`io-fs-media`** | Yes | Yes | The media filesystem. |
| **`iofs-ipod.so`** | Yes | Yes | The iPod driver. |
| **`iofs-ser-ipod.so`** | Yes | No | The serial transport. |
| **`iofs-usb-ipod.so`** | No | Yes | The USB transport. |
| **`iofs-i2c-ipod.so`** | If using chip. | Yes | The i2c interface to the Apple authentication chip. |

See also the *MME Utilities Reference* for detailed information about the relevant drivers and transports.

### Licenses

Special licenses are required to access and use iPod devices. Please contact Apple to obtain the licenses needed for your environment.

## Authenticating iPods

Two methods are available for authenticating iPod devices:

- an authentication chip on your system
- a cross transport authentication chip in the cable connecting the iPods to your system

### Apple authentication chip

An Apple authentication IC chip is required for USB transport connections, and is highly recommended for serial transport connections. An Apple authentication IC chip ensures:

- full functionality of Apple current devices — for example, *without* an authentication IC chip:

- iPods do not have access to video browsing and digital audio
- iPhones report the message: “Accessory unsupported”

- compatibility with future Apple devices

The QNX **`iofs-i2c-ipod.so`** module allows the QNX iPod driver to access the authentication chip using a standard i2c driver. If you do not install the authentication chip as a standard i2c device, you must write a custom module that gives the QNX iPod driver access to the chip. For more information, see “Link kit for iPod authentication” below.

To instruct the iPod driver to use this authentication method, start **`iofs-ipod.so`** with the the **`acp`** option set to **`i2c`** and its options:

```
# io-fs-media -dipod,transport=usb,acp=i2c[:options]
```

### Cross transport authentication

A cross transport authentication chip can be built into the cable connection iPods to your system. This authentication method is available for USB transport connections *only*; it authenticates iPods over the serial pins and tell the iPods to grant authenticated privileges to the USB transport.

As well as offering the same advantages as an authentication chip built in to your system, a cross transport authentication chip in a cable:

- places the authentication chip in a swappable cable rather than on a board
- eliminates the need for the iPod driver to perform authentications

Please contact Apple for more information about licenses and specifications for a cross transport authentication chip.

To instruct the iPod driver to use this authentication method, start **`iofs-ipod.so`** with the the **`acp`** option set to **`cta`**:

```
# io-fs-media -dipod,transport=usb,acp=cta
```

The following iPod devices support cross transport authentication, provided they have the *latest firmware*:

- all iPod touches
- all iPhones
- all iPod nano 4G

## Connecting to iPods

You can connect from a QNX system to any iPod device (including an iPhone) with an Apple 30-pin connector. You can connect to these devices through a QNX serial device (“2-wire” connection) or a QNX USB device (“1-wire” connection).

You can connect to newer iPods (Generation 5 and more recent) either through a serial connection or, with an Apple authentication chip, through a USB connection. Older iPods (Generation 4 and older) support only the serial connection.

Before designing your client application, you should contact:

- Apple to obtain the specifications for cables supporting the serial protocol, and the required authentication IC chip and associated licenses
- QNX for more information about your requirements for iPod devices

iPod nano 2G devices: use a high speed port for connections to these iPods; do not use a full speed port.

The iPod nano 2G refuses to connect if the upstream port reports full speed; this device can not complete a control transfer if the interrupt endpoint is polled before the status phase of the control transfer is complete.

### Accessing iPods as USB devices

Some iPods support USB connections as mass storage devices; when accessed in this way they look like hard drives. Older iPods and iPod Shuffles can be accessed only as USB storage devices:

- iPod Shuffles do not have an Apple 30-pin connector; they use a USB connector and present themselves as USB mass storage devices.
- If an iPod that is accessed as a USB mass storage device (older models and Shuffles) uses a file system that can be mounted onto a QNX system (for example, DOS), the MME can play the iPod’s contents just like it plays the contents of any USB mass storage device. However, the MME cannot access the contents of iPod that use a proprietary Apple file system (iPod’s that have been formatted on an Apple computer).

### Apple devices that support digital audio

At time of this MME release, the following Apple devices supported digital audio:

| Model | Firmware* |
|---|---|
| iPod nano 1G | 1.2.0 |

*continued...*

| Model | Firmware* |
|---|---|
| iPod nano 2G | 1.1.2 |
| iPod nano 3G | 1.0 |
| iPod 5G | 1.2 |
| iPod classic | 1.0 |
| iPod touch | 1.1 |
| iPhone | 1.1 |

*Firmware listed is the minimum required.

A very small number of iPods with outdated firmware may present themselves as supporting digital audio, when in fact they do not support it.

If an iPod falsely presents itself as supporting digital audio, the launcher has no way of telling that the presentation is false. It will launch the digital audio driver, not the driver for a USB storage device, and the MME will be unable to play media or do anything with the iPod.

You should therefore design your client application to detect this sort of situation and alert the user so he or she can intervene and mount the iPod as a USB mass storage device.

### Connecting through a serial device

When you connect to iPod devices through a serial device, the connection uses an iPod-specific protocol.

The iPod-specific protocol is used regardless of the type of physical layer used for the connection. A serial device can be a serial port (such as a 16550), a USB-to-serial class driver, or any other interface that presents a serial device.

For this type of connection, you can manufacture a serial to USB iPod cable to easily connect iPods into any USB port on an existing system. This cable should have an iPod 30-pin connector at one end, a USB connector at the other end, and a USB to serial chip inside the cable.

The iPod driver **`iofs-ipod.so`** and the serial transport **`iofs-ser-ipod.so`** together support the serial protocol required to communicate with iPods.

- Serial connections to iPods are sometimes referred to as “2-wire” connections.
- When you connect to an iPod through a serial device you must route the audio to the appropriate location in the system.

#### Two-wire connections

Serial connections to iPods use two wires:

- The first wire provides a serial interface from the host to the iPod's dock interface. This wire is used to send control over the serial interface. USB to serial converters (**devc-serusb**) may make this interface *appear* to be a USB interface, but it is in fact a serial interface.
- The second wire provides analog audio, just like a headphone jack. This wire can be connected:
  - directly to speakers
  - directly to an amplifier
  - to an **io-audio** managed audio device, which is in turn managed by the MME and **io-media**

  All three configurations for the second wire above are valid.

If you choose to bring the analog audio into **io-audio** and have the MME manage it, you need to:

- run the audio *in* line to a sound card capture device
- use the iPod serial transport (**iofs-ser-ipod.so**) **audio** option to set the URL to the location where **io-media** can read the audio data

If you do not bring the audio into **io-audio**, the MME can receive time position updates and track changes from the iPod through the control lines, but it cannot control volume, mute or other functionality on the iPod.

#### Hardware requirements for serial connections to iPods

When designing a system that will use a serial interface to iPod devices you must ensure that the hardware on your system:

- supports analog audio input
- has sufficient CPU to move data around
- has sufficient CPU to perform sample rate conversion at output, or that hardware and drivers support conversion in the hardware

### Connecting through a USB device

When you connect to iPod devices on a USB bus, the connection uses an iPod-specific USB protocol.

To connect to iPod devices and communicate with them through a USB connection, you need:

- the USB device enumerator: **enum-usb** that presents USB devices to the system

- the iPod driver: **iofs-ipod.so**
- the USB transport: **iofs-usb-ipod.so** that supports the USB protocol required to communicate with iPods
- an Apple authentication chip
- the driver for the Apple authetication chip: **iofs-i2c-ipod.so**, or a custom driver
- the audio driver: **io-audio**
- the audio driver for iPods: **deva-ctrl-ipod.so**

- USB connections to iPods are sometimes referred to as “1-wire” connections.
- When you connect to an iPod through a USB device you must also launch a USB audio driver for the iPod.

#### One-wire connections

USB connections to iPods use one wire:

- The wire used for one-wire iPod connections has a USB connector on the host end and an iPod dock on the iPod end.
- Control packets are sent across this connection as USB **hid** commands.
- The iPod sends audio digital PCM data across the USB connection isochronously, which decodes the music to the host.
- The host uses **io-audio** and **io-media** to handle the PCM data.

### Starting the drivers

To use iPod devices, after you obtained the required licenses, cable and authentication chip, follow the instructions in the *Installation Note* provided with your QNX Aviage Multimedia Suite package, and install the following software components onto your system:

- **deva-ctrl-ipod.so**
- **io-fs-media**
- **iofs-ipod.so**
- **iofs-ser-ipod.so**, for serial connectins
- **iofs-usb-ipod.so**, for USB connections
- **iofs-i2c-ipod.so**, or a custom driver, as required

- **enum-usb**, for USB connections

After you have installed the required components, to connect from a QNX system to an iPod device:

1. Start **io-usb**, specifying the **-c** option and the driver.
2. Start **io-fs-media**, specifying the device, the transport (serial or USB, as required) with either the path to the device or the device name, the authentication chip interface with the address, path and speed for the connection to the chip, other options as required. For example, to connect to an iPod on the default serial port and using the authentication chip, start **io-fs-media** as follows:

   ```
   # io-fs-media -dipod,transport=ser,acp=i2c
   ```

3. If your system is configured to use **io-audio**, start it. See "Starting **io-audio**" below.
4. Physically connect an iPod device to your system.

Below is a sample startup:

```
# io-usb -c -duhci -dehci
# io-audio -dipod busno=0,devno=1,cap_name=ipod-0-1
# io-fs-media -dipod, \
    transport=usb:devno=1:busno=0:sconfig:audio=/dev/snd/ipod-0-1, \
    darates=+8000:11025:12000:16000:22050:24000, \
    playback,acp=i2c
```

- The **io-usb** may use the EHCI, OHCI or UHCI driver, as required.
- The **io-usb -c** option is needed to instruct **io-usb** *not* to select the iPod configuration.
- The **io-fs-media** iPod transport's **sconfig** option selects the USB configuration. You can omit it if you use a launcher that selects the USB configuration.

See the *MME Utilities Reference* for more information about the iPod driver options and default values, and **io-usb** in the *Neutrino Utilities Reference* for more information about **io-usb**.

**io-fs-media** is single threaded, so you need to start a separate instance of the filesystem for each device to which you want to connect.

### Starting `io-audio`

All USB connections and serial connections that route audio through the MME require **`io-audio`**. If you system configuration requires **`io-audio`**, you should start it before physically connecting an iPod device.

When you start **`io-audio`**, you need to specify:

- **`busno`** — the USB bus number
- **`devno`** — the iPod device number
- **`cap_name`** — the name given to the capture device; you must pass the path to the capture device to the **`io-fs`** iPod driver through its **`audio`** option so that **`io-media`** will know where to read the audio data

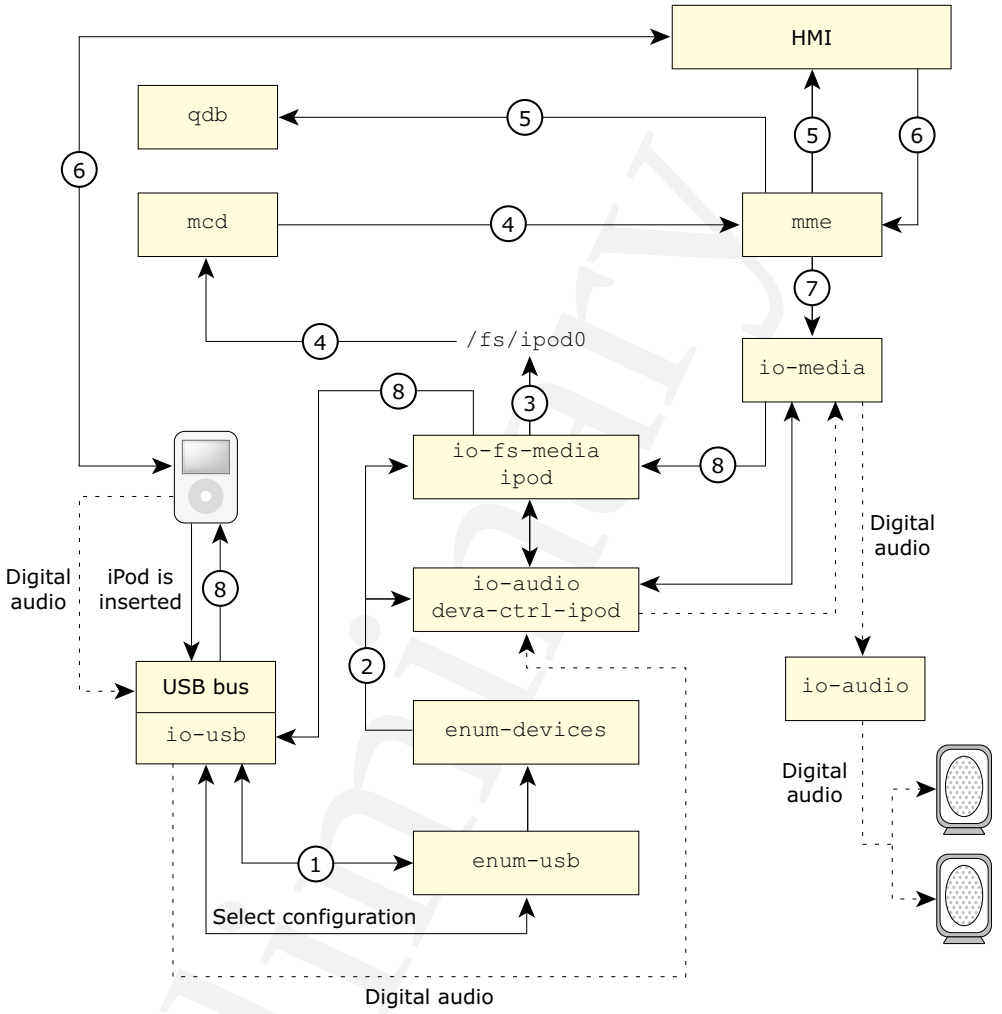
For example:

```
# io-audio -dipod busno=0,devno=1,cap_name=ipod-0-1 &
```

For more information about **`io-audio`** and its options, see **`io-audio`** in the *Neutrino Utilities Reference*.

### iPod one-wire: from connection to playback

The figure below shows the sequence of activities from the insertion of an iPod through a USB connection to playback:




*Activities sequence after iPod connection through USB transport.*

**1** The launcher (**`enum-usb`**) sees the iPod, then sends the device information, such as vendor ID and device ID, to **`enum-devices`**, which uses this information to determine the driver it must launch.

**2** **`enum-devices`** launches **`io-fs-media ipod`** and an **`io-audio`** instance.

**3** **`io-fs-media ipod`** places the iPod in the filesystem: **`/fs/ipod0`**.

**4** **`mcd`** notices the appearance of **`/fs/ipod0`** in the filesystem, and notifies **`mme`**.

**5** **`mme`** writes iPod information in its database through **`qdb`**, and notifies the HMI that the iPod is present.

**6** The HMI explores the iPod, then asks **`mme`** for playback of some tracks on the iPod.

7 **mme** tells **io-media** to start playback on the iPod.

8 **io-media** tells the iPod to start playback, and routes the digital audio from the iPod to the speakers through a second **io-audio** instance.

### Checking for optimal connections

To check it an iPod is using the optimal available connections, you can examine the *capabilities* field for the device in the **mediastores** table:

- Check for the capability MME_MSCAP_CONNECTION_NONOPTIMAL (0x00040000) to determine if the iPod is using a USB (1-wire) or a serial (2-wire) connection for control. MME_MSCAP_CONNECTION_NONOPTIMAL is set if it a serial (non-USB) connection is used for control.
- Check for the capability MME_MSCAP_AUDIO_NONOPTIMAL (0x00080000) to determine if the iPod is using an analog or digital audio connection. MME_MSCAP_AUDIO_NONOPTIMAL is set if an analog audio connection is being used.

The MME_MSCAP_*_NONOPTIMAL flags apply only to iPods that are capable of playback and control via USB connections as well as serial and analog connections. For early iPods models, which only support serial and analog connections, a 2-wire connection is “optimal”, as it is the only one possible.

For a full list of possible mediastore capabilities values, see MME_MSCAP_* in the *MME API Library Reference*.

## Detecting iPods

When the MME detects an iPod, it updates the **mediastores** table just as it does with other types of mediastores. It sets the *storage_type* column in the **mediastores** table to the mediastore type, MME_STORAGETYPE_IPOD, and it updates the **folders** table with the root folder information for the iPod.

After receiving an MME_EVENT_MS_STATECHANGE event with **mme_ms_state_t** indicating a newly inserted mediastore, to check if the mediastore you are working with is an iPod, you can check this column in the **mediastores** table for the presence of an iPod. The **SELECT** statement below shows how to check for an iPod:

```
qdb_statement( db, "SELECT 1 FROM mediastores
                    WHERE msid = %lld AND storage_type = %d;",
                    event.data.msid, MME_STORAGETYPE_IPOD );
```

## Removing iPods

When you have finished playing media from an iPod, you do not need to disconnect it from the MME. Just physically remove it from the system. Note, however, that when an iPod is removed from the system, the MME removes the iPod content from its

database — it maintains an entry for the iPod in the **`mediastores`** table and an entry for the iPod root folder in the **`folders`** table, but the content of the iPod "disappears" from the system.

This MME behavior is specific to iPod devices and is implemented because it isn't possible to quickly determine if any changes were made to an iPod device between its removal from the MME system and its re-insertion: the MME needs to resynchronize the iPod to ensure the accuracy of its data.

## Synchronizing iPods

The MME supports synchronization of media on iPod devices. The design of iPod devices imposes some constraints on how the MME performs synchronizations on iPods. If the default configurations for iPod synchronizations are used:

- The MME never automatically synchronizes an iPod. The client application must request synchronization.
- If the client application requests full, recursive synchronization of all media on the iPod device (by calling *mme_resync_mediastore()*), the MME performs synchronization via the path **`/Music/Genres/`**, then repeats the process via the path **`Music/Songs/`**.

To improve the browsing of iPods, the MME updates the *title* field during the first synchronization pass. This behavior is unique to iPod synchronizations.

To synchronize an iPod, you should use the directed synchronization function *mme_sync_directed()*, specifying the path where you want to begin synchronization. For more information, see "Directed synchronization" in the chapter Synchronizing Media.

We don't recommend that you use the standard synchronization function *mme_resync_mediastore()* to perform a full, recursive synchronization of all media on an iPod device, due to the size of iPod databases, duplicate files on iPods, and the slow interface between the MME and these devices.

### Configuring MME iPod synchronizations

You can configure:

- the maximum number of folders containing files allowed in the MME database by setting the *limit* attribute for the ../**`<ipod>`**/**`<synced_folders>`** option in the MME configuration file
- the MME to automatically synchronize iPods, and to synchronize the complete contents of iPods

For more information, see "Configuring MME iPod synchronization" in the *MME Configuration Guide*.

## Playing media on iPods

An iPod manages its own:

- track sessions
- repeat and random modes

These characteristics place some constraints on what the MME can do with these devices, and they determine client application behavior when working with these devices. Specifically, iPods require special consideration when working with track sessions, when resuming playback, and when using random and repeat modes.

For information about how to check mediastore and device capabilities, see "Mediastore and device capabilities" in the chapter Working with Mediastores.

### Rules for playing media on iPods

To avoid unexpected behavior, follow the rules listed below when playing media on iPods. For more detailed information, see the relevant sections below.

- An MME track session should contain only one *fid* per iPod. If the track session spans multiple iPods, include only one *fid* per iPod.
- To move to the next or previous track in the MME track session (in the **`trksessionview`** table), call the *mme_next()* and *mme_prev()* functions
- To move to the next or previous track in the *iPod* track session, call *mme_button()* with **`mm_button_t`** set to MM_BUTTON_NEXT or MM_BUTTON_PREV, as required.

**CAUTION:** Do not set autopause (*mme_setautopause()*) for control contexts with an iPod. Because iPods control their own playback, if you set autopause for a control context with an iPod:

- playback from the iPod may produce unexpected behavior
- metadata and other track information requested from the iPod may be invalid

### Working with track sessions when using iPod devices

To play media on an iPod, call the same functions as for other devices:

- *mme_newtrksession()* to create the track session
- *mme_settrksession()* to set the track session
- *mme_play()* to start playback

However, because iPods manage their own track sessions, when you use the MME to play media on these devices, you in fact have two layers of track session:

- the MME track session
- the track session on the iPod

These two layers of track sessions mean that, once the MME has started playback on an iPod, unlike devices that do not manage their own track sessions, the iPod continues playback on its own. When it reaches the end of a track, the iPod starts playback of the next entry in the *iPod* track session, and continues until it either comes to the end of its track session, or the client application tells it to stop by calling *mme_stop()*.

This behavior means that if you place multiple file IDs (*fid*s) from an iPod in an MME track session, playback will:

1. Start at the *fid* on the iPod.
2. Continue playback through every track in the iPod's track session before returning to the MME track session.
3. Start at the next *fid* in the MME's track session.
4. If this *fid* is on the iPod, continue from this *fid* through every trackin the iPod's track session, and so on.

For example, if an iPod folder has five files, and you create a track session for an iPod using a **`SELECT`** statement, such as **`SELECT fid FROM library WHERE folderid=x`**, that is appropriate for other devices, you would place all five tracks from the folder (A, B, C, D, E) in the MME track session. Assuming that the iPod repeat and random modes are off, this MME track session would, in fact, play 15 tracks on the iPod, as follows:

```
A, B, C, D, E
B, C, D, E
C, D, E
D, E
E
```

To prevent the situation described above and to avoid unintentional repetition of iPod track sessions, when using iPods, place only one *fid* from the iPod in the MME track session.

The tables below describe playback behavior of two MME track sessions. The first table shows an MME track session with multiple *fid*s from an iPod (not recommended). The second table shows an MME track session with one *fid* per iPod (recommended). The *sequentialid* is from the **`trksessionview`** table; it is the *fid* for playback in sequential mode. Both examples assume that random and repeat modes are off.

**Not recommended: more than 1 *fid* per iPod**

| Mediastore | *sequentialid* | `nowplaying`*fid* | Behavior |
|---|---|---|---|
| HDD | 12 | 12 | 1. Play *fid* 12.<br>2. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| iPod 1 | 16 | 0 | 1. Play *fid* 16 on iPod 1.<br>2. Play all other tracks after *fid* 16 in the iPod track session.<br>3. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| iPod 1 | 17 | 0 | 1. Play *fid* 17 on iPod 1.<br>2. Play all other tracks after *fid* 17 in the iPod track session.<br>3. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| USB 1 | 23 | 23 | 1. Play *fid* on USB 1.<br>2. Move to the next *sequentialid* entry in the **`trksessionview`** table. |

**Recommended: 1 *fid* per iPod**

| Mediastore | *sequentialid* | `nowplaying`*fid* | Behavior |
|---|---|---|---|
| HDD | 12 | 12 | 1. Play *fid* 12.<br>2. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| iPod 1 | 17 | 0 | 1. Play *fid* 17 on iPod 1.<br>2. Play all other tracks after *fid* 17 in the iPod track session.<br>3. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| iPod 2 | 96 | 0 | 1. Play *fid* 96 on iPod 2.<br>2. Play all other tracks after *fid* 96 in the iPod track session.<br>3. Move to the next *sequentialid* entry in the **`trksessionview`** table. |
| USB 1 | 23 | 23 | 1. Play *fid* on USB 1.<br>2. Move to the next *sequentialid* entry in the **`trksessionview`** table. |

- If you created your MME track session to play tracks exclusively from the iPod, it should have only one *fid* and you can start playback by calling *mme_play()* with the *fid* argument set to 0 (zero).
- If your MME track session includes tracks from iPods *and* from other devices and mediastores (CDs, USB sticks, etc.):
  - If you want to play only the tracks on the iPod, you must call *mme_play()* with the *fid* argument set to the *fid* for the track in the iPod track session where you want to start playback.
  - If you want to play all tracks (from the iPod, and from other devices and mediastores) in the track session, you can call *mme_play()* with the *fid* argument set to 0 (zero) to start playback from the first track in the MME track session.

### Getting track information when playing media on iPods

An iPod sends back to the MME metadata such as track title and artist, which the client application can display to the end-user. However, because iPods manage their own track sessions, the MME has no way of knowing information that the iPod doesn't report, such as the filename or the file ID of the currently playing track in the iPod track session. These constraints mean that if playback is on an iPod:

- The MME sets the fid in the nowplaying table to 0.
- The *fid* (**`mme_event_data_t.trackchange.fid`**) delivered with the MME_EVENT_TRACKCHANGE event is the file ID listed in the MME track session (*sequentialid*).
- *mme_play_get_info()* reports the currently playing *fid* listed in the MME track session (*sequentialid*), and the MME_PLAYSUPPORT_* flags. It does not have access to the file ID or file name, etc. in the iPod track session.

If you want to know if a track is playing on the iPod track session, you can:

- Check the MME_PLAYSUPPORT_DEVICE_track session flag to see if the device is an iPod.
- Check the *fid* in the **`nowplaying`** table for 0, which means the track currently playing is on the iPod.

### Getting the time position when playing media on iPods

When playback is on an iPod, the MME reports the playback time position just as it does when playback is on other devices. Note, however, that iPods usually deliver events every 500 milliseconds. If the MME notification interval is less than 500 milliseconds (the default setting is 100 milliseconds), client applications that rely on delivery of time events from an iPod may see jitter in time-position reporting. For more detailed information, see *mme_set_notification_interval()* in the chapter MME API.

## Moving through an iPod track session

To move to the next or previous track in an iPod track session, call the *mme_button()* function with the *button* argument set to MM_BUTTON_NEXT or MM_BUTTON_PREV, as required.

Manage trick play behavior and modes in the same manner as for other devices, with calls to functions such as *mme_play_set_speed()* to fast forward, reverse, pause, etc.

MM_BUTTON_NEXT and MM_BUTTON_PREV are the only *mme_button()* settings supported by iPods.

### Fast forward and reverse on iPods

Fast forward and reverse, and reporting of the current speed is implemented differently on iPods than on most other media devices:

- 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.
- 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.

The **`damping_audio_writer`** filter has no effect on iPods because these devices control their own trick play behavior.

## Resuming playback on iPods

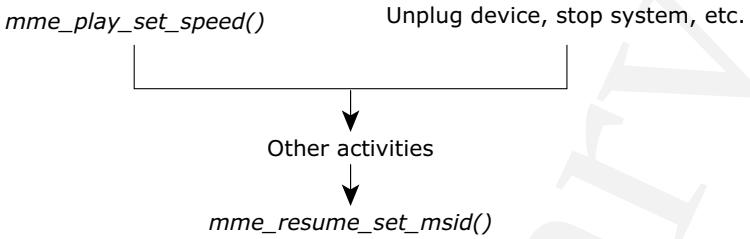
The MME doesn't have access to detailed track session information on devices, such as iPods, that manage their own track sessions. This limitation means that when the MME creates a track session for media one of these devices, it simply:

- passes information, such as play time and metadata, from the device to the client application
- passes commands to the device

If you want to stop, then resume playback of a track session on an iPod, you must:

1. Do whatever action the user requests: pause playback, switch to another activity, note that the device has been disconnected from the system, etc. The device is responsible for saving the state of track session.
2. Call *mme_play_resume_msid()* to resume playback of the track session. This function creates a new MME track session, and the device is responsible for resuming playback from the point where it was stopped.

Calling *mme_play_resume_msid()* when the iPod device itself is in a stopped state will not resume playback, because a stopped iPod has no active track session that can be resumed.




*Stopping and resuming a device-controled track session.*

**CAUTION:** A call to *mme_play()* stops playback after *mme_play_resume_msid()* has been called, because *mme_play()* finds no playable tracks in the MME track session.

The only exception to this behavior occurs when the user has browsed the iPod and requests playback of a specific folder's contents: Artist, Genre, etc. In this case, the client application should create a new MME track session with the *fid* requested by the user, and use *mme_play()* to start playback on that track session.

### Using random and repeat modes on iPods

An iPod maintains its own random and repeat modes. The MME works with this characteristic and behaves as follows with iPod devices:

- Calling the following functions "pushes down" the random and/or repeat modes from the MME track session and sets them on the device:
  - *mme_play()*
  - *mme_setrandom()*
  - *mme_setrepeat()*
- Calling *mme_play_resume_msid()* "pulls up" the repeat and random modes from the device and sets them on the MME control context and track session.

Remember that:

- New track sessions inherit the repeat and random modes from the control contexts in which they are created.
- If you re-use an old track session, this track session keeps its random and repeat modes and passes them to the control context in which it is being used.
- Not all devices accept all random and repeat modes supported by the MME.

- 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.

With all versions (up to 1.1 as of this writing) of the iPhone and iPod touch, the following behavior has been observed:

- A request to set the repeat mode to repeat single takes effect on the next track, not on the currently playing track.
- When repeat single mode is set, it is not possible to turn repeat mode off: the MME repeat mode is turned off as expected, but the iPhone remains in repeat single mode.

To correct this behavior, remove the iPhone or iPod touch, then re-insert it into the system.

### Seeking chapters on iPods

To seek to a chapter on an iPod, simply call *mme_seek_title_chapter()* as you would to seek to a chapter on a DVD-video. When the chapter changes, the MME will deliver the event MME_EVENT_MEDIA_STATUS.

Similarly, to get the number of titles and chapters in the current track, and the currently playing title and chapter numbers, use *mme_get_title_chapter()*.

For more information, see "Starting playback from a specific DVD title and chapter" in the chapter Playing and Managing Video and DVDs.

### Setting subtitles on iPods

To set subtitles on iPods and to get subtitle information from iPods, use the *mme_video_get_subtitle_info()* and *mme_video_set_subtitle()* functions like you would for other devices that support video.

For more information, see "Managing video attributes" in the chapter Playing and Managing Video and DVDs.

## Displaying information from an iPod

The **`iofs-ipod`** presentation alters filesystem names of folders and files. Characters are substituted to allow for leading "." (periods), embedded spaces and so on. In addition, **`iofs-ipod`** adds an index number, prefixed by a "~".

The MME corrects these names when it places them in database columns, such as *title* that are not filesystem names, and when it builds pseudo-metadata for the **`library_genres`**, **`library_artists`**, and **`library_albums`** tables.

Further corrections to how information from an iPod is displayed must be handled by the client application. An iPod uses URL encoding in name strings, so a string such as

"Folk/Rock" will display as "Folk%2FRock" to the end user, unless the client application decodes it.
Common URL encoding strings that need to be decoded from their hexadecimal values before displaying them to the end user are listed in the table below:

| Character | Encoding |
|---|---|
| # | %23 |
| $ | %24 |
| % | %25 |
| & | %26 |
| + | %2B |
| , | %2C |
| / | %2F |
| : | %3A |
| ; | %3B |
| = | %3D |
| ? | %3F |
| @ | %40 |

### Retrieving artwork from iPods

Retrieval of artwork from an iPod is limited to the artwork for the currently playing iPod track *only*.

To retrieve the artwork for the currently playing *iPod* track, use the MME's Load-on-Demand metadata extraction API as you would to retrieve metadata from any other device. That is:

1. Call *mme_mme_metadata_create_session()* to create a metadata session.
2. Use the *mme_metadata_getinfo_current()* function to get the artwork information.
3. Call *mme_metadata_image_load()* to load the artwork.

- The MME supports iPod artwork in color only; it does not support grayscale iPod artwork.
- iPod images are BMP files.
- Due to a hardware limitation of 3G iPods, the MME does not support splash screen loading for these devices.

For more information about metadata sessions for the Load-on-Demand metadata extraction API, see “Getting artwork” in the chapter Metadata and Artwork.

## Uploading splash screens to iPods

You can upload color or greyscale splash screen images to iPod devices when you start the iPod driver **`iofs-ipod.so`**. For more information, see “Splash screens” on the *MME Utilities Reference* page for **`iofs-ipod.so`**.

## HD radio tagging

Client applications that support HD radio tagging for iPods can do so when running the MME. To implement radio tagging:

- Start **`io-fs-media`** with the **`storage`** option to turn on HD radio tagging support; for example: **`# io-fs-media -dipod,storage, ...`**.
- Simply pass instructions directly to **`io-fs-media`**, which passes the instructions on to the iPod. The next time the iPod is plugged into a system with iTunes, iTunes will pick up the radio tags.

HD radio tagging requires authentication.

Below is an example of code that can be used to write HD radio tags to an iPod. Your client application should be designed to manage writing to more than one iPod (**`/ipod0`**, **`/ipod1`**, etc.).

```
#include <sys/dcmd_media.h>

...

if((fd = open("/fs/ipod0/.FS_info./control, O_RDWR)) == -1) {
    printf("Failed to access ipod");
} else {
    rc = devctl(fd, DCMD_MEDIA_IPOD_TAG, plist_single, plist_single_len, &ret);
    if (rc == 0)
        printf("Items were written, only iTunes can confirm contents");
    else
        printf("Write of tag file to iPod failed err:%d (%s): ret:%d", rc, strerror(rc), ret);
}
```

- *plist_single* is the pointer to the XML tag; see the Apple documentation.
- The *devctl() ret* argument returns the number of bytes written to the iPod.

### iPods that support HD radio tagging

At time of this MME release, the following iPod models supported HD radio tagging:

| Model | Firmware* |
|---|---|
| iPod nano 3G | 1.0 |
| iPod 5G | 1.2.3 |
| iPod classic | 1.0 |

*Firmware listed is the minimum required.

# Link kit for iPod authentication

This section describes the MME link kit for building a customized iPod ACP (Authetication CoProcess) module for use with Apple authentication chips.

- About the iPod authentication link kit
- The iPod ACP module
- Using the custom iPod ACP module

## About the iPod authentication link kit

The MME includes the **`iofs-i2c-ipod.so`** module that allows the QNX iPod driver to access the iPod authentication chip using a standard i2c driver. Due to the multitude of possible target board configurations, this module may require modification in order to give the QNX iPod driver access to the Apple authentication chip in your environment.

The MME link kit for iPod authentication is provided to assist you in developing any custom authentication coprocessor communication modules you require. It includes the source code for the sample **`iofs-i2c-ipod.so`** module, which you can use as an example and modify as needed.

## The sample iPod ACP module

The iPod ACP module is a plugin to the iPod driver. It allows the higher-level driver to communicate with the Apple authentication chip without worrying about the hardware specifics of the board on which it is running.

The sample iPod ACP module provided is a generic i2c implementation, which can be modified and implemented with many, different custom transport mechanisms. It contains the complete framework required for iPod ACP modules. You only need to modify its hardware and transport sections to meet the specifications for the ACP you will use.

Before compiling and using the sample iPod ACP module, you must:

- have a board on which you can perform your tests
- install the BSP package you will use (with **`hw/i2c.h`** and relevant drivers)

### iPod ACP module functions

The sample functions provided with the authentication link kit are describe below.You can modify these functions to meet the needs of your environment, maintaining the specified behaviors and return values. For more detailed information about these functions, see the source file, **`acp_i2c.c`**.

Unless otherwise noted, on failure these functions return -1 and set *errno*.

#### *ipod_i2c_addinfo()*

The *ipod_i2c_addinfo()* is useful for debugging. It adds information, as instructed, to the debug/information XML file created when an iPod filesystem is mounted. This function adds any requested informatiom into the XML file's i2c section, or any other section of the file, as required.

#### *ipod_i2c_cpready()*

If the generic iPod driver needs to check the CP_READY signal, it can call the *ipod_i2c_cpready()* function.

This sample function returns the state of the CP_READY signal:

- 0 — not ready
- 1 — ready

If this function is not implemented, it returns ENOSYS

Checking for a CP_READY signal is not required for Apple 2.0 rev B authentication chips; new designs should use this chip or a newer chip.

### *ipod_i2c_init()*

When **io-fs-media** with an iPod driver is started, the iPod ACP module's initialization function is called. This function should:

- parse any command-line options needed for the module
- open and initialize its connection to the Apple authentication chip

The initialization function, *ipod_i2c_init()*, provided with the sample module:

- parses the options for
  - an i2c resource manager path
  - the i2c bus speed
  - the address on the i2c bus
- opens a connection to the i2c resource manager
- sets the necessary bus speed, keeping the connection open

### *ipod_i2c_lock()*

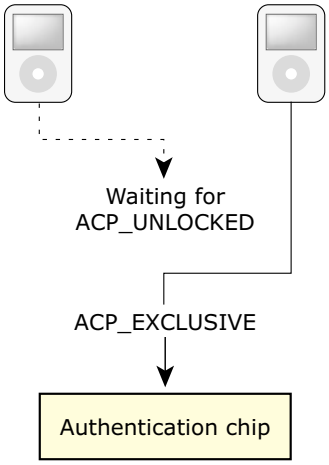
The *ipod_i2c_lock()* locks and unlocks access to an ACP chip so that multiple iPod drivers can run concurrently, sharing access to a single ACP chip. The generic iPod driver for i2c calls this function with a the *lock* parameter set to a value defined by **acplock**.

When *ipod_i2c_lock()* is called with its lock parameter set to ACP_UNLOCKED, it tells the ACP chip that it is not needed until further notice. This call can return ACP_PWROFF, which tells the generic ACP code to power down the ACP chip by writing the appropriate commands to the chip's registers. When the generic code finishes telling the ACP chip to power down, it calls *ipod_i2c_lock()* again, this time passing it ACP_PWROFF to indicate that the chip power down processing has completed.

#### acplock

The enumerated value **acplock** defines the lock settings used by the ACP module. These settings are:

- ACP_UNLOCKED — unlock the ACP
- ACP_SHARED — starting to use the ACP, but only sending atomic commands; no response expected, so an exclusive lock is not required
- ACP_EXCLUSIVE — starting a challenge-response sequence; locking is required to exclude other **io-fs-media ipod** instances until a response is received
- ACP_PWROFF — generic power down processing done




*Two iPods sharing an authenitication chip.*

### *ipod_i2c_read()* and *ipod_i2c_write()*

The *ipod_i2c_read()* and *ipod_i2c_write()* functions read and write data from and to an Apple authentication chip. They return the number of bytes read or written.

The parameters for these functions include:

- the register address on the chip at which to read or write
- a pointer to the buffer for data read in, or with the data to write
- the number of bytes of data to read or written

## Using the iPod ACP module

After you have added all your device-specific code to your custom iPod ACP module, you can build it, then use **`io-fs-media`** to load it.

### Building the module

You can build the iPod ACP module as follows:

1. Copy the sample modules to your home directory:

```
# cp -R $QNX_TARGET/examples ~/examples
```

2. In your home directory, make and install the module:

```
# cd ~/examples/io-fs/drvr/media/ipod/acp/i2c
# make install
```

To build all sample modules, you can make them from the **`~/examples/io-fs`** directory, as follows:

```
# cd ~/examples/io-fs
# make install
```

### Starting the module

You can use **io-fs-media** and the iPod driver to load your module. For example:

```
# io-fs-media -dipod,transport=usb,acp=modulename
```

You can also pass options to the module, as follows:

```
# io-fs-media -dipod,transport=usb, \
    acp=acp_modulename:path=/sample/path:option2=someoption
```

For a list of options available with the ACP sample module, see **iofs-i2c-ipod.so**.

# Chapter 15

# Working with PFS Devices

## *In this chapter. . .*

This chapter describes how to use the MME to synchronize and play media on PFS-enabled devices:

- Installing MME components for external media players
- Directed PFS device startup
- Detecting and synchronizing PFS devices
- Playing media on PFS devices
- Devices that don't support **`GetPartialObject`**

See also **`iofs-pfs.so`** in the *MME Utilities Reference*, and User-specified MTP commands to PFS devices in the *MME Technotes*.

# Installing MME components for external media players

If you want use an external media player, such as an iPod or a PlaysForSure-enabled device, you need to:

1. Install the runtime files that support these devices. These installations may require special licenses.
2. Use **`iofs-ipod`** or **`iofs-pfs`**, depending on the type of media player.

For more detailed instructions, see the QNX® Aviage Multimedia Suite *Installation Note*.

# Directed PFS device startup

The PFS driver used by the MME can be started with one program instance per PFS device, rather than with a single program instance servicing multiple PFS devices. You have the option of starting **`iofs-pfs`** to service multiple PFS devices or to support one PFS device per instance of **`iofs-pfs`**.

To start **`iofs-pfs`** to support one PFS device per instance of **`iofs-pfs`**, use the **`device`** option and specify the paths for the bus, device and interface for each. For example, to handle two PFS devices (**`device=`***bus_no*`:`*device_no*`:`*interface_no*):

```
# io-fs-media -dpfs,device=1:3:3
# io-fs-media -dpfs,device=2:4:6
```

Bus, device and interface numbers are hexadecimal values.

# Detecting and synchronizing PFS devices

When the MME detects a PFS device, it updates the **`mediastores`** table just as it does with other types of mediastores, and sets the *storage_type* column for the mediastore to MME_STORAGETYPE_MEDIAFS. To check if the mediastore you are

working with is a PFS device, simply check the value of this column for the mediastore's entry in the **mediastores** table.

## Optimizing PFS device synchronization

PFS playlist album files (with the extension **.alb**) list album contents and can slow synchronization of PFS devices. To optimize synchronization of PFS devices, you should use the **<SyncFileMask>** in the MME configuration file **mme.conf** to instruct the MME to skip files with the extension **.alb**:

```
<Configuration>
    ...
    <Database>
        ...
        <Synchronization>
        ...
            <SyncFileMask>\.alb$</SyncFileMask>
        ...
        <Synchronization>
    <Database>
<Configuration>
```

The **<SyncFileMask>** element can define multiple character strings identifying files to be ignored by the MME synchronization. For more information, see "Configurable file skipping: **<SyncFileMask>**" in the *MME Configuration Guide*.

# Playing media on PFS devices

This section describes considerations specific to playing media on PFS devices, and to getting artwork from these devices. In includes:

- Playing DRM content
- Decryption of DRM content

## Playing DRM content

The MME supports PlaysForSure (PFS) devices that play DRM-protected media. To play DRM-protected media on PFS devices, you must obtain the required key files from Microsoft, and use the MME's PFS module (**iofs-pfs**).

If you will use DRM-enabled devices on your system, you should start the PFS module with the **drm** option, so that it checks for the required files and exits if it does not find them. This strategy ensures that the MME does not fail when it attempts to playback DRM-protected media. For more information about configuring the MME to support PFS devices playing DRM-protected media, see "Configuring Digital Rights Management (DRM)" in the *MME Configuration Guide*.

When you have finished playing media from a PFS device, you do not need to disconnect it from the MME. Just physically remove it from the system.

### Decryption of DRM content

DRM content decryption uses an AES block cipher with a 128-bit key. The AES key is unique to each playback session; it is different every time a song is played.

When the PFS module registers itself with the PlaysForSure device, it sends a certificate. This certificate contains the 1024-bit RSA public key that the device will use to encrypt the seed used to determine the AES key.

When the user selects DRM-protected content:

1. The PFS module requests a license for that content from the PlaysForSure device.
2. The device returns the license, which includes an encrypted seed.
3. The PFS module uses a private RSA key to decrypt the seed, then uses the decrypted seed to determine the key for the AES block cipher.
4. The PFS module processes every 128 bits of encrypted content with the 128-bit AES key to yield the decrypted content for playback.

## Devices that don't support `GetPartialObject`

The default configuration for **`io-fs-media`** is to support only PFS devices that implement the **`GetPartialObject`** MTP command, as specified by the PFS 2.01 specification. Unless **`io-fs-media`** is configured to support PFS devices that don't support the **`GetPartialObject`** MTP command, attempting to access such a devices produces an **`info.xml`** file with the following: **`<NotSupported>`**101b**`</NotSupported>`**.

If your environment requires that you use devices that don't support the **`GetPartialObject`** MTP command, you must specify the **`getsize`** option and the buffer size when you start **`io-fs-media`**. For example, to specify a 3 megabyte buffer:

```
# io-fs-media -dpfs,getsize=3M
```

Devices that support only the **`GetObject`** MTP command require that the MME have enough memory allocated by the **`getsize`** option for it to read in the entire file. If the file exceeds the memory allocated, the read fails. Note, however, that you can have the MME allocate memory dynamically by setting **`getsize`** to 0 (**`getsize`**=0). This configuration allows the MME to use as much memory as is available to read a file, allowing it to read in bigger files — with the danger that all available dynamic memory may be allocated for a file read, leaving no dynamic memory available for other uses.

For more information about how to configure **`io-fs-media`** to work with legacy devices that don't fully support the PFS 2.01 specification, see **`iofs-pfs.so`**.

# Chapter 16

# Working with Bluetooth Devices

## *In this chapter. . .*

The MME Bluetooth devices. This section describes how to integrate Bluetooth devices into the MME, and provides some basic information about accessing Bluetooth device functionality through the MME:

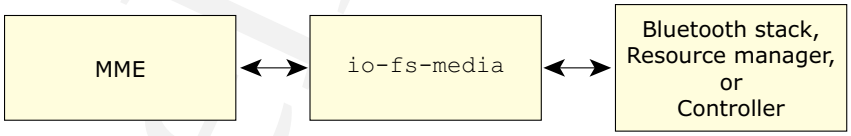
- Integrating Bluetooth audio devices into the MME
- Creating a Bluetooth device representation to MediaFS specification
- The **io-fs-media** module example
- Modifying the **io-fs-media** module example
- Using the **io-fs-media** module example
- Messages for controlling Bluetooth devices
- Using Bluetooth devices with the MME

For information about how to get configuration values from a Bluetooth device, see "Getting and setting external device options" in the chapter External Devices, CD Changers and Streamed Media.

# Integrating Bluetooth audio devices into the MME

The MME uses the **io-fs-media** interface to control playback on remote devices. This interface permits device agnostic playback control and metadata extraction, and lets the MME control playback without detailed knowledge of the underlying device.

To control Bluetooth audio devices, the MME uses the MediaFS interface provided by **io-fs-media**.




*The MME uses the* **io-fs** *interface to control a Bluetooth audio device.*

See also **io-fs-media** in the *MME Utilities Reference*.

To use Bluetooth devices with the MME you must integrate them into the MME. This integration requires three tasks:

1. Create a Bluetooth device representation to MediaFS specification.
2. Configure the MME for Bluetooth support.
3. Control the Bluetooth device through the MME and play media.

You should be familiar with the QNX *MediaFS (Media File System) Specification* before starting work on integrating Bluetooth devices into the MME. If you do not have the latest MediaFS specification, contact your QNX representative.

# Creating a Bluetooth device representation to the MediaFS specification

The Aviage Bluetooth Integration Kit provides the resource manager framework required for representing Bluetooth devices via **`io-fs-media`**. This resource manager provides the ability to load user-created modules that uses the MediaFS specification to describe devices.

Access to a control device is specific to that device and depends on how the device is represented by the system. The access interface can be via any of:

- a Bluetooth stack
- a resource manager
- a memory mapped controller

You can also create your own resource manager to represent Bluetooth devices, following the MediaFS specification.

# The `io-fs-media` module example

To facilitate development of new **`io-fs-media`** AVRCP modules describing Bluetooth devices, the Aviage Bluetooth Integration Kit includes an **`io-fs-media`** module example that implements the Bluetooth AVRCP (Audio/Video Remote Control Profile).

This example contains the complete framework required for **`io-fs-media`** modules that describe devices. You only need to modify device-specific sections to change the module so that it can access your required underlying control devices.

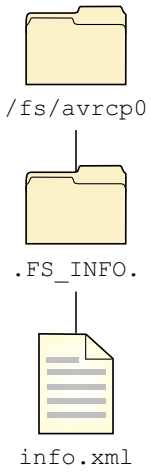
## What the `io-fs-media` module example does

The **`io-fs-media`** AVRCP module is a plugin to the **`io-fs-media`** resource manager. When **`io-fs-media`** with an AVRCP module is started, it:

- calls the code entry point *avrcp_mount()*
- starts a timer
- waits to be informed that a remote device is present

When it learns of a remote device, the **`io-fs-media`** resource manager registers a path, called the *mountpoint*, to the location of MediaFS filesystem. The default

mountpoint used by the module provided with the Aviage Bluetooth Intergration Kit is **/fs/avrcp0**. When you have configured the MCD, it will monitor the system for this path. See "Modifying the MCD configuration file for Bluetooth" below.




*The **io-fs-media** MediaFS module hierarchy*

Since the **io-fs-media** AVRCP module implements only a subset of the MediaFS filesystem hierarchy, the only item at this path is the **.FS_info.** directory.

The **.FS_info.** directory contains MediaFS entries used for playback control and device information extraction. The directory for the **io-fs-media** AVRCP module contains only the **info.xml** file. The first request to read the **info.xml** file causes *avrcp_getinfo()* function to populate the file's contents.

All playback and metadata extraction messages are issued through an open file descriptor to the **info.xml** file. These are handled by the function *avrcp_devctl()*. This function and other major functions in **avrcpexample.c** are described below. For more detailed information about these and other functions in **avrcpexample.c**, see the source file.

## avrcp_devctl()

The *avrcp_devctl()* function is the workhorse for all device control messages. It translates MediaFS playback and metadata extraction commands, then transmits them to the remote device. All commands handled by the *avrcp_devctl()* function require the addition of device-specific control code to the **io-fs-media** module example.

See "Configuring the MME for Bluetooth support" for more information about available commands.

## avrcp_mount()

The *avrcp_mount()* function is the entrance point for the **io-fs-media** module. It allocates memory associated to the driver. After allocating memory and registering

internal functions, *avrcp_mount()* enables a timer that handles polling for remote device insertions and removals.

You can modify the structure **avrcp_device** defined in the **avrcpexample.h** header file to store user data that persists for a mount period.

### *avrcp_options()*

The *avrcp_options()* function passes arguments to the driver at the **io-fs-media** interface. The **io-fs-media** module example includes the following set of options:

- **device** — the control device to which commands are issued
- **mount** — specify the mountpoint
- **verbose** — a value to enable logging at verbosity levels; the function *fs_log()* can use the verbosity level to filter logging.

### *avrcp_timer()*

The function *avrcp_timer()* is used to detect the presence of a remote device to mount and register its path. To use this function you must add device-specific code to the **io-fs-media** module that will communicate the presence of a remote device so that it can be mounted.

Because the mountpoint for the **io-fs-media** module can be registered and unregistered, if a remote device is present and available to play, *avrcp_timer()* registers its mount path. If the remote device is removed from the system, *avrcp_timer()* unregisters the mount path.

## The mount process

The mount process has two stages:

1. Establish a valid connection to the resource manager or AVRCP controller.
2. Confirm that the remote device is connected.

If the resource manager or the AVRCP controller is unavailable, the **io-fs-media** module uses the *avrcp_timer()* function's "poll" until it detects that the required resource is available for mounting.

The poll rate is set by the user in the configuration options. See above.

## The **avrcpexample.h** header file

The **avrcpexample.h** defines the structure **avrcp_device** and several constants used by the **io-fs-media** AVRCP module.

```
#include <inttypes.h>
#include "media.h"

#define AVRCP_NAME        "AVRCP"
```

```
#define NAME_BUF_SIZE     512
#define AVCP_METADATA_MAX 1024

struct avrcp_device {
    struct mediafsdevice  mediafs;
    char *                devpath;
    int                   fd;
    char                  dname[AVCP_METADATA_MAX];
    char                  dserial[AVCP_METADATA_MAX];
    uint16_t              verbose;
};
```

The structure **avrcp_device** is the principle AVCP device structure. It is defined in the **avrcpexample.h** header file and carries information about these devices. It can be extended to hold data about an underlying device. Its standard members include:

| Member | Type | Description |
|---|---|---|
| *mediafs* | **structmediafsdevice** | An opaque structure; it must be present and first. |
| *devpath* | **char** | The pathname to the AVCP device resource manager. |
| *fd* | **int** | The file descriptor connection to device resource manager, or -1 if not connected |
| *dname*[AVCP_METADATA_MAX] | **char** | Bluetooth-friendly name |
| *dserial*[AVCP_METADATA_MAX] | **char** | Bluetooth address (used as device serial number) |
| *verbose* | **uint16_t** | The log level verbosity |

# Modifying the io-fs-media module example

This section describes how to modify and build the **io-fs-media** module example.

## Adding device-specific code to the module

The Aviage Bluetooth Integration Kit includes a set of files that make up the **io-fs-media** module. These files include:

- **avrcpexample.c** — the source code for the module
- **avrcpexample.h** — a header file with the **avrcp_device** structure

The only file that requires changes for integration is the **avrcpexample.c** file. Sections of this file that require the addition of device-specific code are denoted by comments, as follows:

- **// -- START DEVICECODE** — the start of device-specific code section

- **// -- TASK:** — a description of what the device-specific code needs to accomplish
- **// -- PSEUDOCODE** — an optional a high-level description of an algorithm or a routine that must be replace by device-specific code
- **// -- EXAMPLECODE**. — optional code that can remain and be used in the device-specific code
- **// -- END DEVICECODE** — the end of device-specific code section

Below is an example of a section of the **avrcpexample.c** file that requires device-specific code:

```
case DCMD_MEDIA_PREV_TRACK:
    // -- START DEVICECODE
    // -- TASK: Issue command to skip to the previous track on
    //          the remote device.
    //          Device control may be supported. If not supported,
    //          status=ENOTSUP.
    //          Device control can fail.
    // -- PSEUDOCODE:
    //    status = DEVICEFUNC_PREV_TRACK();
    // -- END DEVICECODE
break;
```

## Building the module

After you have added the required device-specific code to the **io-fs-media** module, you can build it as follows:

1. Copy the sample modules to your home directory:

   ```
   # cp -R $QNX_TARGET/examples ~/examples
   ```

2. In your home directory, make and install the module:

   ```
   # cd ~/examples/io-fs/drvr/media/avrcpexample
   # make install
   ```

To build all sample modules, you can make them from the **~/examples/io-fs** directory, as follows:

```
# cd ~/examples/io-fs
# make install
```

# Using the io-fs-media module

After you have added all your device-specific code to the **io-fs-media** module, you can use **io-fs-media** to load it. Starting the module is as simple as running an instance of **io-fs-media** with the driver, as follows:

```
# io-fs-media -davrcpexample
```

You can also pass options to the module, as follows:

```
# io-fs-media -davrcpexample,verbose=10,mount=/fs/alt/mount/point
```

See *avrcp_options()* for a list of options.

If the remote device is present, then your application should register a new mount point. You can use the **ls** commandline instruction to examine the mountpoint:

```
# ls /fs/avrcp0
```

You can add multiple devices by running separate instances of the module, as follows:

```
# io-fs-media -davrcpexample,dev=/dev/avrcp0,mount=/fs/avrcp0
# io-fs-media -davrcpexample,dev=/dev/avrcp1,mount=/fs/avrcp1
# io-fs-media -davrcpexample,dev=/dev/avrcp2,mount=/fs/avrcp2
```

# Messages for controlling Bluetooth devices

This section lists the messages that you can send to Bluetooth devices to control playback and to extract metadata.

- Remote devices differ, so device-specific control codes may not support some commands. If a command is not required and is not supported, it must return an error with *errno* set to ENOTSUP.
- All state modification control messages must be *synchronous*. A requested action must either complete or fail before returning. For example, if the state modifier DCMD_MEDIA_PLAY message is issued, upon return of the *devctl()* call, the underlying device must be in a playing state, or have returned a POSIX error indicating why the command failed.
- For more information about commands, see the MediaFS specification.

## Playback messages

Playback of media tracks on a Bluetooth device occurs on the device. The start and manage playback the **io-fs-media** module sends control messages to the Bluetooth device. The table below lists playback commands implemented for AVRCP 1.0 and 1.3 devices. Required commands are marked with an asterisk (*). All others are optional.

### AVRCP 1.0

The **io-fs-media** driver implements the following playback **devctl**s for AVRCP 1.0 devices:

- DCMD_MEDIA_PLAY*

- DCMD_MEDIA_PAUSE*
- DCMD_MEDIA_RESUME*
- DCMD_MEDIA_NEXT_TRACK
- DCMD_MEDIA_PREV_TRACK
- DCMD_MEDIA_FASTFWD
- DCMD_MEDIA_FASTRWD
- DCMD_MEDIA_PLAYBACK_INFO*

#### AVRCP 1.3

In addition to the playback commands listed above, the following optional shuffle and repeat commands **`devctl`**s are implemented for AVRCP 1.3 devices:

- DCMD_MEDIA_GET_SHUFFLE
- DCMD_MEDIA_SET_SHUFFLE
- DCMD_MEDIA_GET_REPEAT
- DCMD_MEDIA_SET_REPEAT

### Metadata messages

Metadata extraction commands retrieve metadata about the currently playing file. All strings returned by the metadata extraction commands must be UTF-8 encoded. All metadata extraction commands are optional.

- DCMD_MEDIA_SONG
- DCMD_MEDIA_ALBUM
- DCMD_MEDIA_ARTIST
- DCMD_MEDIA_GENRE
- DCMD_MEDIA_DURATION
- DCMD_MEDIA_TRACK_NUM

## Using Bluetooth devices with the MME

This section describes:

- how to configure the MME for Bluetooth support
- how to set up an MME track session on manage playback on a Bluetooth device

## Configuring the MME for Bluetooth support

After you have modified your **io-fs-media** module and have it running, you must configure the MME to enable Bluetooth support. You need to:

- modify the MCD configuration file
- add an entry for Bluetooth to the **slots** table

### Modifying the MCD configuration file for Bluetooth

To enable Bluetooth support, add an entry to the MCD configuration file to instruct the MCD to monitor the path for Bluetooth devices. For example:

```
[/fs/avrcp*]
Callout     = PATH_MEDIA_PROCMGR
Argument    = /proc/mount
Priority    = 11,10
Start Rule  = INSERTED
Stop Rule   = EJECTED
```

For more information about the MCD, see the *MME Utilities Reference*.

### Enabling Bluetooth support in the slots table

The Bluetooth slot type is MME_SLOTTYPE_MEDIAFS (4), and the storagetype is MME_STORAGETYPE_A2DP (12). To enable Bluetooth support, you need to add an entry such as the following in the **slots** table:

```
INSERT INTO slots(path,zoneid, name, slottype)
    VALUES('/fs/avrcp0', 1, 'Bluetooth', 4);
```

The *slottype* for Bluetooth support must be **4**.

## Playing media on Bluetooth devices

If you have configured the MME correctly to support Bluetooth devices, on learning of the insertion of a Bluetooth device, the MME:

- creates an entry for the device in the **mediastores** table
- inserts a single **FTYPE_DEVICE** file ID (*fid*) into the **library** table

You can use this file ID to create a track session with a single track for the device, then issue commands to start and manage playback. Playback remains on the Bluetooth device, and the MME does not have access to information about an individual track on the device unless the track is being played.

### MME playback features supported for Bluetooth devices

The MME supports calls to the following playback functions for Bluetooth devices:

- *mme_stop()*
- *mme_play()*
- *mme_play_set_speed(0)*
- *mme_play_set_speed*(1000)
- *mme_button(MM_BUTTON_NEXT)*
- *mme_button(MM_BUTTON_PREV)*

**CAUTION:** Do not set autopause (*mme_setautopause()*) for control contexts with a Bluetooth phone. Because Bluetooth phones control their own playback, if you set autopause for a control context with a Bluetooth phone:

- playback from the device may produce unexpected behavior
- metadata and other track information requested from the device may be invalid

Below are sample sequences showing how to use the MME commandline utility (**`mmecli`**)to:

- query the MME **`mediastores`** table for a Bluetooth device
- query the MME **`library`** table for the file ID for the Bluetooth device
- create a track session and start playback on the Bluetooth device

#### Query the `mediastores` table

```
# qdbc -d mme "select msid,slotid,storage_type, mountpath \
    from mediastores"
Rows: 2  Cols: 4
Names:  +msid+slotid+storage_type+mountpath+
00000:  |1|1|2|/media/drive|
00001:  |2|9|2|/fs/avrcp0|
```

#### Query the `library` table

```
# qdbc -d mme "select fid,msid,folderid,ftype,filename, title \
    from library where msid=2"
Rows: 1  Cols: 6
Names:  +fid+msid+folderid+ftype+filename+title+
00000:  |2|2|2|5||Bluetooth|
```

### Create a track session and start playback on the Bluetooth device

```
# mmecli newtrksession l "Select fid from library where msid=2"
(rc=0,errno=0) new trksessionid=2.  Execution Time=0.010

# mmecli settrksession 2
(rc=0,errno=0) Set trksessionid=2.  Execution Time=0.031

# mmecli play
(rc=0,errno=i0) Playing from tracksession fid = 2.  Execution Time=0.038
```

## Getting metadata

The MME has access to metadata for a track on a Bluetooth device only when the device is playing the track. To obtain this metadata, query the MME's **nowplaying** table.

## Audio routing

The MME does not handle audio routing for the **io-fs-media** module. You must configure your system to properly route audio from Bluetooth devices to the desired output location.

# *Chapter 17*

# MME Sample Applications

The standard MME package includes several sample applications, which provide simple examples of how to perform basic tasks with the MME. These sample applications are:

- **`mme-shuffle`** — just play some tracks

The MME also includes the source code for the command-line utilities:

- **`mmecli`** — issue commands corresponding to MME API calls

## Syntax:

**mme-shuffle [-d** *mme_device***] [-r]**

## Runs on:

Any platform that supports Photon and the MME.

## Options:

| | |
|---|---|
| **-d** *mme_device* | Use the *mme_device* to connect to the MME. By default, **mme-shuffle** uses **/dev/mme/default** |
| **-r** | Set the playback mode to random playback. The default mode is sequential mode. |

## Description:

The sample application **mme-shuffle** demonstrates how to use the MME in a simple way. It performs all required operations to connect to the MME, create a new track session, implementing play, stop, pause, continue, next and previous operations.

## Examples:

See the application source code.

## See also:

**mme-player-simple**, **mme-cdripper**, **mme-player-multisource**

# *Glossary*

### Checkpoint

A snapshot of a database, usually RAM-based, that is copied to persistent storage, such as a hard drive or flash. The checkpoint can be used to restore the database after a power cycle or if the database becomes corrupt.

### CBR

Constant bitrate.

### Control context

A multimedia output point, or location where media files can be played. A control context represents an audio output device, can hold a single track session, and can play a single track at a time. By default, the MME has one control context, but you can add more to the MME database, then connect to them. An MME client receives notifications from an attached control context.

### CPPM key

Content Protection for Pre-recorded Media — a key used for DRM.

### Codec

Coder-Decoder — an program that encodes and/or decodes a digital data stream or signal.

### DSP

Digital Signal Processor — a microprocessor that processes digital signals is real-time.

### DTS

Digital Theater System — a multi-channel digital sound format.

### DRM

Digital Rights Management — a generic term for technologies used to control access to and usage of copyrighted works. The MME supports files protected with Microsoft Windows Media DRM, via the **`iofs-pfs.so`** PFS module to **`io-fs`**.

### FID or *fid*

File ID — in the MME, a unique identifier for media files and tracks

### File

In the context of the MME, “file” refers to all non-media files (the MME configuration file, for instance) and to media files that are being read or otherwise manipulated for a purpose other than playing them. See also “track” below.

### Locale code

The locale code is a string containing a 5-character language and region code. This code 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`**

### Media

Any music, pictures, or video, in block or stream format.

### Mediastore

Any source for multimedia data; examples include hard drives, DVDs, CDs, and media devices such as an iPod or MP3 player.

### Metadata

Data describing a media track. Metadata includes but is not limited to track name, artist(s), release date, genre and so on.

### MME

Multimedia Engine — the Aviage Multimedia Suite's multimedia engine.

### MTP

Media Transport Protocol — a protocol developed by Microsoft for synchronizing both protected and unprotected media content on portable media devices such as MP3 players.

### Playback

The act of playing of a media track.

### Playlist

A list of media files (identified by FIDs). Your application can play a playlist by creating a track session from it.

### PlaysForSure (PFS)

A certification given by Microsoft to media devices that use the MTP protocol.

### Prune management

A technique of ensuring that a database doesn't grow too large or exceed a specified size by "pruning" (deleting) unused data.

### Ripping

Ripping is the process of reading files from a mediastore, changing the format of these files into another format if required, then writing the files in their new format to a mediastore or other storage device. Copying media is simply ripping media and writing the destination file in the same format as the source file.

### Synchronization

The process by which the MME examines mediastores and updates its database with information about the media tracks on the stores and with the metadata for these media. Information and metadata includes but is not limited to media type and format (audio, video, etc.), track name and language, genre and cover art.

### Track

In the context of the MME, "track" refers to media files that are being played or read and otherwise accessed or manipulated for playing. For example, the MME synchronizes folders and the *files* inside them, but it reads the *tracks* from a playlist and places them in a track session. See also "file" above.

### Track session

A list of media tracks (identified by FIDs) that can be played by the MME on a specific control context.

### Trick play

The term "trick play" refers to playback operations (such as fast forward, reverse and skip) that are not straightforward, normal speed playback.

### UOP

User Operation Prohibitions — prohibitions placed on what users can do when manipulating a video.

### VBR

Variable bitrate.

### Zone

In the MME, an area to which output devices are attached, and to which the output from media playback is sent.

# Index

## B



## C

## D

## E



## F



## G

## H



## I

## J



## L



## M

## N



## O



## P

## Q



## R

## S

## T



## U

## V



## W



## Z