Multimedia Test Utilities Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 31, 2015

Contents

About This Guide

The *Multimedia Test Utilities Guide* describes the multimedia test and demo tools that help developers prototype, test, and debug media apps. The binaries of some of these utilities are included in the installer package, so you can run them from the command line. For other utilities, their source code is included in the platform's source code package to provide a programming reference for using multimedia services.

To find out about:	Go to:
The capabilities of the multimedia test utilities and their role in the development lifecycle	*Role of the Multimedia Test Utilities* (p. 9)
How to test the APIs of other multimedia components by using **mmcli**	*mmcli* (p. 13)
How to play media files through **mm-renderer** by issuing **mmcli** commands	*Using mmcli to play media files* (p. 23)
How to play media files with single **mmrplay** commands	*mmrplay* (p. 25)
How to configure and run **mm-pnp** to demonstrate the accessing, extracting, and playing of mediastore content	*mm-pnp* (p. 35)

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`
Constants	NULL
Data types	**unsigned short**
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

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 typically use a forward slash (`/`) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Role of the Multimedia Test Utilities

The QNX® SDK for Apps and Media includes several multimedia utilities to help you rapidly prototype, test, and debug media apps.

These utilities let you issue basic commands to play tracks and upload media information to databases without having to manage communication sessions with multimedia services or manually define inputs and outputs. The binaries of **mmcli** and **mmrplay** are included in the platform installer, which unpackages them on your host system. You can obtain the source code for **mmrplay** and **mm-pnp** by downloading the platform's source code package, which is found on the same webpage as the installer.

Using the multimedia test and demo tools, you can:

- learn about the media information flow between the device publishers, the QDB database server, the **mm-sync** service, and the **mm-renderer** service
- adapt the code in the multimedia plug-and-play program (**mm-pnp**) to customize the process of detecting mediastores and then synchronizing and playing their media content
- play an audio or video file through **mm-renderer** by issuing a single command
- exercise the APIs of the multimedia services and libraries included with the product

When you're writing early versions of media apps and you need to support only common use cases, it's convenient to use the multimedia test utilities because you need to know relatively few commands. The device publishers and the **QDB**, **mm-sync**, and **mm-renderer** services all have more complicated command interfaces and require more time to learn and start using. These services should be used in later development stages to support more complex use cases and to optimize performance and the user experience.

Multimedia test utilities

The QNX SDK for Apps and Media includes these multimedia test utilities:

Utility	Description	Binary in installer	Source code in platform source code package
mmcli	Tests the APIs of multimedia components by forwarding commands from a script file or an interactive session to a loaded library or service.	Yes	
mmrplay	Plays or records media through **mm-renderer**, after configuring the service based on command-line options.	Yes	Yes
mmsyncclient	Forwards media synchronization commands to **mm-sync** and reports synchronization status. This utility is described in the *Multimedia Synchronizer Developer's Guide*.	Yes	Yes
mm-pnp	Multimedia plug-and-play tool, which provides a source code walkthrough of detecting a device, synchronizing its metadata, and playing its media files. You can modify the configuration file to customize how the tool responds when certain devices are attached.		Yes
charconvert2_icu	Character converter plugin, which can be used for internationalization.		Yes
dbnotifydemo	Monitors database changes made by **mm-sync**, by using **libdbnotify**.		Yes

The source code package for the platform is available at the same download location as the installer.

Using the multimedia test utilities

The test and demo tools are command-line utilities that don't have their own APIs. Instead, users can enter high-level commands in a QNX Neutrino terminal to automate common media-management tasks. This design saves you from having to link in client-side libraries (for C programs) or use JavaScript extensions (for HTML5 apps) to integrate your prototype apps with multimedia services.

You can display a list of supported options for each tool with the `use` system command, as shown in the following example for **mmcli**:

```
% use mmcli

mmcli [options] [cmdfile]...

Options:
 -i DLL [options] -- load interface DLL
 -l log_file
 -D name=value    -- define a global variable
 -h file          -- save command history to that file
 -H               -- save command history to "cli.history"
 -L               -- display time as hh:mm:ss instead of seconds
                     since start
 -v               -- increase verbosity of CLI and all loaded
                     modules
```

Chapter 2 mmcli

The **mmcli** utility forwards commands to multimedia components so that you can test their APIs. These commands can be listed in files or entered in an interactive session.

You must start **mmcli** with a command line that lists any desired logging options as well as the files containing the media commands that you want to execute (we refer to these files as *test scripts*). The **mmcli** tool configures itself based on the options given and then executes the test scripts. Then, **mmcli** starts an interaction session, which lets you enter media commands and see their results. If you don't name any test scripts when starting **mmcli**, the utility just starts the interactive session and waits for media commands.

With **mmcli**, you can test the APIs of these multimedia components:

- **mm-renderer**, for managing playback of single tracks or playlists
- **libmd**, for extracting metadata from files on mediastores
- **libmmplaylist**, for navigating playlists and retrieving their tracks

To use the **mm-renderer** API in **mmcli**, you must be running the **mm-renderer** server process. For instructions on starting **mm-renderer**, see the *Multimedia Renderer Developer's Guide*.

Using mmcli instead of other components

The **mmcli** utility helps you to learn the APIs of multimedia libraries and services and to test their functions without having to integrate those multimedia components with your media apps.

With **mmcli**, you can start learning and using the multimedia components before you begin developping the apps that will use them. This learning activity can help you in your initial design of media apps and your subsequent implementation of media operations within those apps.

When to use mmcli

You should use **mmcli** if you want to:

- exercise the APIs of multimedia components in an interactive command session
- write and execute test scripts to assess the functionality and performance of multimedia components
- log all media commands and their results so that you can review the command history and analyze any errors

When to use other components directly

You shouldn't use **mmcli**, but instead directly call the APIs of multimedia components if you want to:

- write production-level media apps that don't depend on a helper utility to forward commands to lower-level services
- optimize media operations by avoiding the **mmcli** overhead of parsing input text and of logging errors
- customize how you handle errors or events reported by a particular library or service

mmcli command line

*Start **mmcli** to execute test scripts and to accept interactive media commands*

Synopsis:

```
mmcli [-D name=value] [-H] [-h history_file] [-i DLL [options]] [-L]
      [-l log_file] [-v] [script_file] ...
```

Options:

-D *name=value*

Define a global variable. The string given for this option must contain a name-value pair, with the name and value separated by an equal sign (=).

Global variables are visible to all test scripts and in the interactive session.

-H

Write the command history to the default history file (**cli.history**). Commands are written to this file in the same order that they're executed, whether they're read from test scripts or entered in the interactive session.

The history file is different from the log file, which you can set with the `-l` option.

-h *history_file*

Write the command history to the specified file. As with the `-H` option, commands are written in the order that they're executed.

-i *DLL [options]*

Load the dynamic library of a multimedia component. The API functions of that component become accessible to **mmcli**.

The library filename can be followed by a set of parameters for initializing the library. You must specify the parameters in a comma-separated list of name-value pairs, in which the name and value in each parameter substring are separated by an equal sign (=). For example, you can load the **mm-renderer** library and tell it to connect to the service named "car" and to create a context named "voice" by setting the following option: `-i mmrenderer_cli.so connect=car,context=voice`

You can provide as many `-i` options as you like to support all the APIs you plan to use. The available configuration options vary with the library.

-L

When logging, display time as `hh:mm:ss.aaa`, where `aaa` is milliseconds. By default, **mmcli** displays time as the number of seconds and milliseconds since it was started, in the format `sss.aaa`.

`-l` ***log_file***

Write the commands issued to **mmcli** and the results of commands to the specified file in addition to displaying this information in the interactive session.

The log file is different from the history file, which you can set with the `-h` option. The log file provides more information because it contains not only the command history but also the results of media operations.

`-v`

Enable verbose mode to log additional information. This option is disabled by default.

`script_file`

Execute a test script of media commands. To ensure **mmcli** can support the commands called by the script, you must either load the necessary library files on the command line (see the `-i` option for more information) or use the *load* command in the script before calling any API functions.

You can name as many test scripts as you like and **mmcli** will execute them in the order listed. This is important to remember if you have dependencies between scripts.

Description:

The **mmcli** utility loads the libraries of multimedia components so that you can call their API functions in test scripts or in an interactive session. You can load certain libraries on startup by specifying one or more `-i` options on the command line. For any of these options, you can list name-value pairs of parameters to initialize the library.

You can log the command history (with the `-H` or `-h` option) and log the results of commands (with the `-l` option). These options allow you to generate and review the complete chronological sequence of media operations with event timing and other information included. The command history and the results of commands are always displayed in the interactive session, even if you've disabled logging to files.

When started, **mmcli**:

1. examines its command-line settings to configure its logging features and to load the libraries of multimedia components (as requested)
2. executes the test scripts named on its command line, in the order listed
3. starts an interactive session and waits for you to enter media commands

Note that you don't have to name any test scripts; in that case, **mmcli** simply skips the second step. If you do provide test scripts to **mmcli**, you must load all library files necessary to support the API commands called by those test scripts. While **mmcli** can ignore or recover from some command specification errors, it can't guess at which multimedia component implements a particular command and then load that component.

The **mmcli** interactive session stays active until you enter `quit` to explicitly exit the program.

mmcli language

In addition to supporting the API commands of loaded multimedia components, **mmcli** has built-in commands that allow you to define and structure sequences of API commands.

The **mmcli** utility parses its input line by line to extract commands. The input text can come from test scripts or an interactive session—**mmcli** processes commands from both sources identically.

In this sense, **mmcli** acts like an interpreter for a scripting language. This language includes a group of built-in functions or *keywords* and it can be extended with the APIs of multimedia components. For commands that correspond to API calls, **mmcli** forwards them to the appropriate components for execution. The keyword commands control the execution of other commands. For example, there are keywords for branching, looping, and common functions such as waiting for an event or passing a command string to the OS. You can use keywords both in test scripts and in the interactive session.

Keywords

The **mmcli** utility supports these keywords:

Keyword	Arguments	Description
`cilist`		List all loaded modules (libraries). The module names are listed three per line, in fixed-width columns.
`clock`	`on	off`	Enable or disable the logging of the current time when writing command history or event information to files. By default, this setting is enabled. If you change it in one test script, this setting remains in effect for all other test scripts until it's explicitly changed.
`echo`	`[text]`	Print out a line of text.
`else`		Start the "else" part of a conditional block. This command must be specified after an `if`, `ifdef`, or `ifndef` command but before its corresponding `fi` command.
`expect`	`n ev_list`	Wait for an event for up to `n` milliseconds. In `ev_list`, you can list the numbers corresponding to the event types that you want to monitor. List entries must be separated with either plus signs (+) or commas. If you don't list any events, **mmcli** monitors all events.
`fi`		End a conditional block. This command must be specified after an `if`, `ifdef`, or `ifndef` command, and that command must not already have another `fi`.
`file`	`script`	Read and execute commands from a test script. The `script` argument contains the test script filename.
`global`	`name val`	Define a global variable. This variable is visible in all test scripts and in the interactive session.
`help`	`[cmd]`	Explain all currently available commands or the command named in `cmd` (if this argument is given).

Keyword	Arguments	Description
		The available commands include all keyword commands and the API commands of any loaded module.
`if`	`ok | error`	Start a conditional block that will be entered depending on the result of the last command. When `ok` is given as the argument, the commands that follow will be executed only if the last command succeeded. When `error` is given, the commands that follow will be executed only if the last command failed. Note that you can nest blocks.
`ifdef`	`cond`	Start a conditional block that will be entered only if the test specified by `cond` evaluates to true. Note that you can nest blocks.
`ifndef`	`cond`	Start a conditional block that will be entered only if the test specified by `cond` evaluates to false. Note that you can nest blocks.
`list`		List all available API functions. The function names and signatures are listed three per line, in fixed-width columns.
`load`	`module`	Load a module (library) to make its API functions available. The `module` argument contains the library filename.
`local`	`name val`	Define a local variable. This variable is visible only in the current test script or the current interactive session.
`log`	`file`	Open a file for logging the results of media commands.
`loop`		End a repeat block. **This command is supported only in scripts.** The commands before a `loop` command but after the last `repeat` command will be executed `n` times (where `n` is specified by the `repeat` command).
`nolog`		Close the log file. Command results will no longer be logged.
`quit`		Exit from **mmcli**. In interactive mode, this command exits the program. In test scripts, this command causes **mmcli** to stop executing the current script and exit altogether, which is useful if a serious error occurs and you want to shut down the program. Note that other scripts named later on the command line won't be executed because **mmcli** will exit immediately.
`repeat`	`n`	Repeat a block of commands. **This command is supported only in scripts.** The commands between a `repeat` command and the next `loop` command will be executed `n` times.
`setdelay`	`n`	Set a delay of `n` milliseconds between executing commands in test scripts.
`shell`	`[cmd]`	Launch a shell and execute a command in that shell (if the `cmd` argument is given). In test scripts, you can examine the return code of the `shell` command to see if the OS successfully executed the command specified in `cmd`. Launching a shell without passing it a command is impractical in a test script because you can't manually enter commands and view their results.

Keyword	Arguments	Description
		In interactive mode, the standard output and error streams from the shell are redirected to **mmcli**, allowing you to view the command results. If you don't provide the `cmd` argument, **mmcli** starts a shell process in the foreground so that you can enter and execute commands in that shell until you close it (by typing `exit`).
`sleep`	`n`	Delay for `n` milliseconds.
`trap`	`on | off`	Set or unset the "trap" condition. If this condition is set, **mmcli** stops processing the current test script if a command fails. By default, this setting is enabled. If you disable it in one test script, the trap condition is re-enabled when **mmcli** begins processing the next test script.
`verbosity`	`module level`	Set the logging level for a module (library). The `module` argument contains the library filename. The `level` argument is either `0` (for "off") or `1` (for "on"). By default, logging is off.

mmcli test scripts

You can automate multimedia testing by executing test scripts with **mmcli**. Test scripts offer a convenient way of rapidly issuing media commands and logging their results.

For **mmcli**, test scripts are simply text files that list one command per line. Each command invokes either a built-in **mmcli** function or an API function from a multimedia component. The script writer is responsible for learning the proper syntax (i.e., the command name and the required arguments) for each command and correctly specifying it; **mmcli** doesn't correct your input.

To execute a test script, you must either provide its filename on the command line or execute it in interactive mode with the *file* command. Unlike interpreters for languages such as Python or Perl, the **mmcli** utility can't be named in the first line of a script to allow the file to be executed from a QNX Neutrino terminal.

Test script example

The following sample test script loads the Multimedia Playlist Library (**libmmplaylist**), opens a playlist file, and fetches the playlist entries up to a maximum number of entries:

```
trap off

load mm-cli-mmplaylist.so

if error
echo Error loading command interface module.
quit
fi

mmplaylist_open / %playlist
```

```
if error
echo Error opening playlist:
echo %playlist
quit
fi

repeat %size
mmplaylist_entry_next_get 1
if error
echo Error retrieving playlist entry.
quit
fi
loop

echo Playlist read complete. Exiting.
quit
```

The *if* and *fi* keywords define error-handling branches, which just print out a brief error message and exit.

Notice the references to the `playlist` and `size` variables in the arguments of the *mmplaylist_open* and *repeat* commands. The percent sign (`%`) in front of a variable name indicates a variable reference. For the script to work, you must define these two variables on the command line (with the `-D` option for each variable) as follows:

```
# mmcli -Dplaylist=/tmp/music/pl/playlists_all/M3Uv1_test.m3u \
        -Dsize=10 /Users/dcarson/work/temp/misc/branching.cli
```

Alternatively, you can instruct **mmcli** to load the playlist library before it runs the script. This way, you don't have to use *load* in the script. To preload the library, use the `-i` command-line option, as follows:

```
# mmcli -i mm-cli-mmplaylist.so \
        -Dplaylist=/tmp/music/pl/playlists_all/M3Uv1_test.m3u \
        -Dsize=10 /Users/dcarson/work/temp/misc/branching.cli
```

The *repeat* and *loop* keywords define the beginning and end of an iterative code section. In this case, *mmplaylist_entry_next_get* is called repeatedly to get playlist entries until either the script fetches the number of entries specified in `size` or it encounters an error. The argument to this API function is the playlist handle; its value is hardcoded to `1` because we know that this is the first (and only) playlist handle created in the script.

The *quit* command on the last line causes **mmcli** to exit instead of going into interactive mode. You can remove this last *quit* command to keep **mmcli** active so that you can enter commands interactively after running the script.

mmcli interactive sessions

After executing any test scripts named on its command line, **mmcli** starts an interactive session that allows you to enter media commands and see their results.

The interactive session is helpful for learning built-in **mmcli** commands (keywords) as well as multimedia API commands because you can use the *help* command to list all available commands. These include the keywords and the APIs of any loaded components. The session also provides tab completion of partially specified command names and allows you to navigate the list of previous commands using the up and down arrow keys.

Some commands, notably *repeat* and *loop*, work only in test scripts. You can't use them in interactive mode.

Interactive session example

The following excerpt from an interactive session shows how to use the Metadata Provider Library (**libmd**) to read metadata from an MP3 file:

```
# mmcli
Starting...
> load mm-cli-libmd.so
Unable to load DLL "mm-cli-libmd.so": Library cannot be found.
005 Command 'load' failed (-1, errno 2); No such file or directory
> load mm-cli-md.so
> mmmd_session_open /fs/usb0
010 Command 'mmmd_session_open':
sessionID: 1
> mmmd_get 1 /tmp/music/tracks/one.mp3 md_title::artist,album,\
                genre,name,comment,duration,width,height,bitrate,\
                samplerate,mediatype 0
054 Command 'mmmd_get':
metadata: md_title_artist::U2
md_title_album::Achtung Baby
md_title_genre::Rock
md_title_name::One
md_title_comment::Download from http://www.last.fm/music/U2
md_title_duration::276522
md_title_bitrate::128000
md_title_samplerate::44100
md_title_mediatype::2147483652
> mmmd_session_close 1
> quit
#
```

Because the first *load* command specifies an incorrect library name, **mmcli** outputs an error message. The second command names the correct library, so **mmcli** loads the API defined in the library and doesn't output anything to the interactive session.

The argument to *mmmd_session_open* is the mountpoint of the device that metadata is being extracted from (in this case, a USB stick). The command outputs the numeric ID of the new session, which must be used as the first argument in the *mmmd_get* call. The second argument to *mmmd_get* provides the path of the track whose metadata we're extracting. Alternatively, you could use the *local* command at an earlier point in the script to set this variable locally and then reference this variable in the command argument by using the percent sign (`%`) in front of the variable name. The third argument lists the metadata fields that we're requesting. The fourth argument must be `0` to tell **libmd** that there's no preferred plugin for extracting the metadata.

The *mmmd_get* command outputs the names and values of all metadata fields that it successfully read. The duration (track length) is given in milliseconds while the bit rate and sample rate are given in Hertz (Hz). The `width` and `height` fields don't apply to audio tracks, so no metadata is retrieved for these fields.

Finally, the session ID is given again in the call to *mmmd_session_close*, which doesn't produce any output.

It's not necessary to call *mmmd_init* and *mmmd_terminate* as the first and last **libmd** functions because **mmcli** does this for you.

Using mmcli to play media files

The following **mmcli** interactive sessions demonstrate common use cases for playing media files with **mm-renderer**. The command sequences shown here don't correspond exactly to the API calls needed if you're using **mm-renderer** directly, but do show all the configuration steps needed to define a media flow, configure parameters, and control playback.

Playing an audio file

To start playing an audio file located on a USB stick over the default audio device, enter an **mmcli** command sequence like this:

```
# mmcli -i mmrenderer_cli.so

> OutputAttach audio:default audio
018.965 Command 'OutputAttach':
ID: 0
> InputAttach /fs/usb0/tunes/arcade_fire/sprawl(II).wma track
> Play
```

The `OutputAttach` command defines an audio output for **mm-renderer** by specifying a URL of `audio:default` and an output type of `audio`. This URL tells **mm-renderer** to use automated audio routing with the Audio Manager service. Because you can define many outputs, **mm-renderer** returns an output ID (which is shown on the following line) for each attached output, so you can distinguish it from the other outputs. You must attach all outputs before attaching the input because **mm-renderer** sometimes determines whether it can play an input based on the types of the attached outputs.

Next, the `InputAttach` command provides a URL with the path of a WMA file stored on a device mounted to **/fs/usb0**, and also specifies the `track` input type (because the input is a single track to be played in isolation). The `Play` command then starts playback.

In this first example, the **mmrenderer_cli.so** library is loaded using `-i` when **mmcli** is launched. This ensures that the **mm-renderer** API is available from the start of the interactive session. You could also load the library just after launching **mmcli**, as follows:

```
# mmcli

> load mmrenderer_cli.so
> OutputAttach audio:default
...
```

When an audio track is playing, you can adjust its speed (with `SpeedSet`), seek to a new position (with `Seek`), and stop playback (with `Stop`). For instance, after starting playback with the previous command sequence, you can give the following commands:

```
> SpeedSet 0
> Seek 30000
> SpeedSet 1000
> Stop
```

This sequence pauses playback (by setting its speed to `0`), seeks to a new position 30 seconds from the track start (as indicated by the `30000` value, which specifies a track offset in milliseconds), resumes playback at normal speed, and finally stops playback. It's not necessary to pause playback before seeking to a new position; this was done just as an example of playback control.

Playing a video file

To play a video file stored on the local hard drive, enter a command set like this:

```
# mmcli -i mmrenderer_cli.so

> OutputAttach screen: video
018.965 Command 'OutputAttach':
ID: 0
> OutputAttach snd:/dev/snd/pcmPreferredp audio
018.965 Command 'OutputAttach':
ID: 1
> OutputParameters 1 volume=30
> InputAttach /tmp/video/abilodeau_gold_medal_run.mp4 track
> Play
```

Here, two outputs are attached—one for the video component, which is rendered by the Screen windowing service, and another for the audio component, which is outputted over the preferred audio device (e.g., a speaker). The `video` or `audio` output type must be given when calling `OutputAttach` for each component. Notice that each output is given a distinct ID, which is displayed on the following line. The volume of the audio output (whose ID is `1`) is set to `30` with the `OutputParameters` command. Next, the `InputAttach` command provides an input URL containing the path of a video file found in the **/tmp/video/** directory. Finally, the `Play` command starts playing the video and audio components.

Playing a different media file

When you've finished playing one file and want to play another one, you must detach the current input and output and then attach a new output before attaching a new input:

```
> InputDetach
> OutputDetach 0
> OutputDetach 1
> OutputAttach
018.965 Command 'OutputAttach':
ID: 2
> InputAttach /tmp/audio/killers/when_you_were_young.mp3
```

The `InputDetach` command requires no parameters because only one input can be set but `OutputDetach` requires the ID of the output being detached. You must call `OutputDetach` for each attached output. After giving these commands, you can define a new media flow by attaching one or more new outputs and then the new input.

Chapter 3 mmrplay

The **mmrplay** utility plays media files by invoking **mm-renderer** to manage the media flow from an input to an output. The utility reports any playback errors and also reports when the media finishes playing (if configured to do so).

The **mm-renderer** service must be running for **mmrplay** to work. For information on starting **mm-renderer**, see “Starting the multimedia renderer” in the *Multimedia Renderer Developer's Guide*.

You can play a media file by providing **mmrplay** with a single command that contains valid URLs for an input and an output. The input URL can refer to:

- the path of a media file in the local filesystem
- an HTTP source (including an HTTP Live Streaming broadcast)
- an audio capture device (microphone)
- an SQL query run on a local database

The **mmrplay** utility accepts the same types of input URLs as **mm-renderer**. For details on the supported input URLs, see the *mmr_input_attach()* function described in the **mm-renderer** documentation.

To find the path of a media file stored on a connected device (e.g., a USB stick), you must know the mountpoint assigned to the device when you attached it to your system. The mountpoint and filesystem information of connected devices can be read from the PPS objects stored in **/pps/qnx/mount/**, which are updated by the *device publishers* when you attach or detach mediastores. These services run constantly in the background, so the platform automatically mounts mediastore filesystems. The *Device Publishers Developer's Guide* provides full information on how the publishers work, including how they publish device mountpoints in PPS.

The output URL can name an audio or video device (for playback) or a file (for recording, which works only for audio content). The output URL types supported by **mm-renderer** (and **mmrplay**) are described in the *mmr_output_attach()* function, also found in the **mm-renderer** documentation.

The **mmrplay** command may also contain:

- the name of the **mm-renderer** service to connect to
- a context name and associated set of parameters
- input parameters (e.g., `repeat`) and the input type (e.g., `track`)
- output parameters (e.g., `volume`)
- logging and verbosity settings

Using mmrplay instead of mm-renderer

Although it conveniently abstracts the many steps required to play media through **mm-renderer**, **mmrplay** is limited in terms of the media operations that it supports.

The **mmrplay** utility can only play a media track or playlist and report any errors and (optionally) the playback completion. It can't accept any commands during playback to select another input or to change the playback position or speed. To support these more advanced operations, you must either use **mm-renderer** directly or load and then call its API in **mmcli**.

When to use mmrplay

You should use **mmrplay** if you want to:

- rapidly and easily prototype a media app by enabling only basic playback, which lets you focus more on designing other app features
- play media files by issuing a single command instead of the lengthy API call sequence necessary to connect to **mm-renderer**, configure an output and an input, start playback, and process and report events

When to use mm-renderer

You shouldn't use **mmrplay**, but instead use **mm-renderer** if you want to:

- offer the ability to pause or stop playback, to adjust the play speed, or to change playlists without interrupting playback
- reuse the same context for multiple playback operations so that you don't have to redefine the context parameters
- support configuration of individual tracks in a playlist
- customize how playback events and errors are handled and reported

mmrplay command line

*Play a media file through **mm-renderer***

Synopsis:

```
mmrplay [-r connectpath]
        [-c contextname] [-m mode] [-C ctxt_param=val]
        [ [-a audio_url] | [-v video_url] | [-f file_url] ]
        [ [-A audio_param=val] | [-V video_param=val] | [-F file_param=val] ]
        [-t inputtype] [-I input_param=val] [-D | -Q] [ -S] input_url
```

Options:

-A ***audio_param=val***

Apply a parameter for audio output. The string given for this option must contain a name-value pair for an output parameter applicable to audio output, with the name and value separated by an equal sign (=). For the list of audio output parameters, see the *mmr_output_parameters()* function in the **mm-renderer** API description.

By default, no parameters are set for audio output. You can name as many output parameters as you need, but each must go in its own `-A` option.

-a ***audio_url***

Name the URL of an audio device to use for output. The URL formats that are valid for audio output are given in the *mmr_output_attach()* section of the *Multimedia Renderer Developer's Guide*.

This option can be specified only once in the command line and shouldn't be used if you're using one of the `-v` and `-f` options.

-C ***ctxt_param=val***

Apply a parameter to the context. The string given for this option must contain a name-value pair for a valid context parameter, with the name and value separated by an equal sign (=). For the list of context parameters, see the *mmr_context_parameters()* function in the **mm-renderer** API description.

By default, no parameters are applied to the context. You can name as many context parameters as you need, but each must go in its own `-C` option.

-c ***contextname***

Name the new context that will manage the media flow for this playback operation. The default context name is "`testplayer`".

-D

Enable additional event logging. When this option is set, **mmrplay** logs messages when the playback finishes and when the input is detached. This option is handy for knowing whether playback completed successfully.

-F ***file_param=val***

Apply a parameter for file output. The string given for this option must contain a name-value pair for an output parameter applicable to file output, with the name and value separated by an equal sign (=). See *mmr_output_parameters()* for the list of file output parameters.

By default, no parameters are set for file output. You can name as many output parameters as you need, but each must go in its own `-F` option.

-f ***file_url***

Name the URL of a file to use for recording media content. See *mmr_output_attach()* for the list of URL formats that are valid for file output.

This option can be specified only once in the command line and shouldn't be used if you're using one of the `-a` and `-v` options.

-I ***input_param=val***

Apply a parameter to the input. The string given for this option must contain a name-value pair for a valid input parameter, with the name and value separated by an equal sign (=). For the list of input parameters, see the *mmr_input_parameters()* function in the **mm-renderer** API description.

By default, no parameters are applied to the input. You can name as many input parameters as you need, but each must go in its own `-I` option.

-m ***mode***

Set the file permission flags for the directory to be used by the new context. By default, the directory has the same read and execute permissions as the user.

-Q

Disable event logging except for errors and warnings. By default, **mmrplay** logs information not only for errors and warnings but also for updates to track metadata or to the playlist window. This option is handy for minimizing the logging while still seeing serious errors.

-r ***connectpath***

Name the **mm-renderer** service to connect to. If you don't provide this option, the default service is used.

-S

Remain open after the end of video playback until explicitly signalled. When this option is enabled, the last rendered screen will remain visible until the client issues either the `slay` or **Ctrl–C** command.

-t ***inputtype***

Set the input type. Acceptable values are: `track`, `playlist`, and `autolist`. The default setting is `track`.

-V ***video_param=val***

Apply a parameter for video output. The string given for this option must contain a name-value pair for an output parameter applicable to video output, with the name and value separated by an equal sign (=). By default, no parameters are set for video output. You can name as many output parameters as you need, but each must go in its own `-V` option.

For video outputs that use the `screen:` URL type, the recommended way to set window properties is either in the URL itself (when attaching the output) or through the Screen library (after you've attached the output). Currently, no useful properties can be set by defining output parameters with `-V`, but this option could be used if new parameters for `screen:` URLs or new URL types for video output are defined in future releases.

`-v` ***video_url***

Name the URL of a video device to use for output. See *mmr_output_attach()* for the list of URL formats that are valid for video output.

This option can be specified only once in the command line and shouldn't be used if you're using one of the `-a` and `-f` options.

`input_url`

Name the URL of the input source. This parameter is required because **mmrplay** must be told what to play or record. The input URL is passed to **mm-renderer**, which supports these input sources:

- files in the local filesystem, including those on attached mediastores mounted in the filesystem
- HTTP streams, including live broadcasts
- audio capture devices (microphones)
- query results from SQL-driven databases

For the list of valid URL formats for each input type, see the *mmr_input_attach()* function in the **mm-renderer** documentation.

Description:

The **mmrplay** utility plays media files by invoking **mm-renderer** to manage the media flow from the specified input to the specified output. You can name only one input in the command line, but this input can be a playlist if you want to play multiple tracks in sequence.

The output that you name can be an audio or video device or a file. If you provide URLs for more than one output type on the command line, **mmrplay** selects the URL to use as the media stream destination in the following order (from most to least preferred): file, video, and audio.

For each input you want to play, you need to provide only one command to **mmrplay** and the utility then configures and uses **mm-renderer** to play the specified input. Each **mmrplay** command opens a new connection and creates a new context with **mm-renderer** to start the playback. When the playback finishes, **mmrplay** detaches the input and closes the connection.

Note that **mmrplay** detects only basic command-line errors, such as invalid options, and doesn't validate the values given for recognized options. Instead, **mmrplay** forwards these option values to **mm-renderer**, which validates them. The **mmrplay** utility displays any parameter errors reported by **mm-renderer** to standard error.

By default, **mmrplay** logs information for:

- errors and warnings related to playback
- metadata updates for a playlist entry
- changes to the playlist window (e.g., the playlist position moved forward by one track, causing another track to enter the playlist range)

When provided with the `-v` option, **mmrplay** logs additional messages when the playback finishes (which indicate whether it finished normally or abnormally) and when subsequently detaching the input and closing the connection to **mm-renderer**. The `-Q` option enables logging for only error and warning events.

The **mmrplay** utility is purely a command-line tool; it has no client library exposing an API in C. Once started, **mmrplay** runs as a self-contained process that doesn't require any user input or accept any commands. You can't cancel the playback or change its speed or position once it's started, and you can't save the input and output URLs or their parameters for subsequent operations. This last design point means that each **mmrplay** command must specify all the parameters to use for that particular playback operation.

Playback examples

The following **mmrplay** commands demonstrate common use cases of playing or recording media files. The media files accessed here are encoded in popular formats such as MP3, M4A, and WMA, but all **mmrplay** commands have the same syntax and support the same options, regardless of the media file format.

In these examples, the input URLs are pathnames in the local filesystem. These can be the paths of files that you uploaded or files stored on connected devices. When you attach a mediastore (e.g., a USB stick, an audio CD) to your system, the device publisher that monitors devices of the same type as the newly attached device will publish its mountpoint to a PPS object in **/pps/qnx/mount/**. For instance, when you plug in a USB stick, the **usblauncher** service assigns it a mountpoint of **/fs/usb0** (or something similar). For information on how the device publishers assign mountpoints and publish device information, see the *Device Publishers Developer's Guide*.

The **mmrplay** playback commands automate the tasks of attaching outputs and inputs and configuring their parameters, which must be done in separate **mm-renderer** API calls before you can play media files. For examples of all the **mm-renderer** actions needed to configure and control the playback of media files, see "*Using mmcli to play media files* (p. 23)".

Playing an audio file

To play an MP3 audio file located on an attached USB stick over the default audio device, enter a command like this:

```
# mmrplay /fs/usb0/stillness_in_time.mp3
```

This command doesn't name an audio output URL using `-a` because **mmrplay** will output the media file to the default audio device if you don't specify an output URL. In this case, **mmrplay** sends an output URL of `audio:default` to **mm-renderer**, which tells it to use automated audio routing with the Audio Manager service. If **mm-renderer** doesn't accept this URL, **mmrplay** then sends it the device pathname of the preferred audio device (e.g., **/dev/snd/pcmPreferredp**). If **mm-renderer** can't use this device either, the playback command fails.

You can set the volume by defining it as an audio output parameter with the `-A` option. You can also explicitly state that the input type is a track (not a playlist) by using the `-t` option:

```
# mmrplay -A volume=30 -t track /fs/usb0/stillness_in_time.mp3
```

The output device is independent of the input format, so you would use the same command syntax to play other types of audio files (e.g., WMA, WAV), except that your input URL would contain a filepath with a different extension.

Playing a playlist

The command syntax for playing the tracks in a playlist is similar to that for playing an individual track, except that you must state the playlist input type (using `-t`) and you can also set the `repeat` input parameter (using `-I`):

```
# mmrplay -t playlist -I repeat=none /fs/usb0/u2_best_of_80s.m3u
```

This command sets the repeat mode to `none` to prevent **mm-renderer** from continuously playing either a single track or the entire set of tracks in the playlist. If you set a different repeat mode (e.g., `all`), **mmrplay** would initiate playback in **mm-renderer** but then wait indefinitely for the playback to stop (because it would never receive an **mm-renderer** event indicating that playback has stopped). In this case, you would have to use the **mm-renderer** API or **mmcli** to explicitly stop playback in the context created by **mmrplay** when you issued the playback command. Stopping playback in a context requires knowing its name, which can be set with `-c`:

```
# mmrplay -c test_playlist_looping -t playlist -I repeat=all \
          /fs/usb0/u2_best_of_80s.m3u
```

In this case, **mmrplay** creates a context called `test_playlist_looping` in **mm-renderer** and then initiates continuous playback of the entire track set in the **u2_best_of_80s.m3u** playlist. To stop playback in a client application or **mmcli**, you would have to connect to **mm-renderer** (using *mmr_connect()*), open the existing `test_playlist_looping` context (using *mmr_context_open()*), and then stop playback in that context (using *mmr_stop()*). For full details on connecting to **mm-renderer** and working with contexts, see the *Multimedia Renderer Developer's Guide*.

Playing a video file

To play an MP4 video file, enter a command like this:

```
# mmrplay -v screen: /fs/usb0/seven_days_live.mp4
```

Here, the video output URL has a prefix of `screen:` to tell **mm-renderer** to output the video stream to the Screen windowing service, which renders it to the display. You can set some properties of the output window in this type of URL, as explained in *mmr_output_attach()*.

If you want to play video without any audio output, you can add the `-a` option and specify an empty URL:

```
# mmrplay -a "" -v screen: /fs/usb0/seven_days_live.mp4
```

Recording audio content

To record audio content to a file, enter a command like this:

```
# mmrplay -c test_audio_recording -f /tmp/my_karaoke.wav \
          snd:/dev/snd/pcmPreferredc?frate=44100&nchan=2
```

This command directs the audio output to a WAV file named with `-f`. Here, the input URL names the preferred audio capture device and provides parameters to set the sampling rate (`frate`) and the number of channels (`nchan`). You can find a list of all supported parameters for this URL type in the *mmr_input_attach()* function description.

The `snd:` input URL type works only with file output, so you must provide a file output URL when using this input URL type.

Similar to playing a playlist with the repeat mode enabled, recording audio content causes **mmrplay** to initiate playback with **mm-renderer** but to never process an event indicating playback has stopped. If you start recording audio with an **mmrplay** command like the one shown here, you must then explicitly stop the recording by stopping playback in the `test_audio_recording` context, by either directly calling the **mm-renderer** API or using **mmcli**.

Chapter 4
mm-pnp

The multimedia plug-and-play tool, **mm-pnp**, is a demo program that exposes the process of detecting a mediastore, uploading its media information to a database, and playing its media files.

This demo program provides a helpful reference for writing your own media apps because it performs all the common media-management tasks. However, the **mm-pnp** program doesn't directly access media content or manage media streams. Instead, it uses other platform services to monitor which mediastores (i.e., devices) are attached and to synchronize and play their media files. You can customize how **mm-pnp** synchronizes and plays content for different device types (e.g., iPods, USB sticks, CDs) by modifying its configuration file.

Source code

The source code for this program is included in the platform's source code samples package, which you can download from the same location as the installer. Note that the appropriate makefiles are also included in the samples package, so you can build the program with the **gcc** compiler on a development system.

The **mm-pnp** code provides a walkthrough of the API call sequences that detect when the user attaches a mediastore and then access, extract, and play its content. The program is written in C and contains just over 3000 lines of code, including whitespace, comments, and preprocessor directives. The code is organized into the following modules:

plug and play

Implements the main thread, which parses the command-line options and starts and stops child threads.

config

Defines default values for all configuration options and parses the configuration file to extract and store all user-specified options.

PPS monitor

Implements two child threads. One thread monitors Persistent Publish/Subscribe (PPS) mount objects to detect when the user attaches a device. Meanwhile, the other thread monitors QDB status objects to respond appropriately when media databases are loaded or unloaded.

DB manager

Provides a callback mechanism so that you can define your own responses to database status changes.

sync

Stores the lists of pending and active synchronizations and communicates with **mm-sync**, which performs the synchronizations. Also, implements the child thread that handles **mm-sync** events.

playback

Stores the list of playlists and communicates with **mm-renderer**, which plays those playlists.

linked list

Defines data types used in lists and implements the functions for iterating through lists and for adding and removing elements.

logging

Stores the verbosity level and the list of active logging destinations (e.g., **sloginfo**, **stdout**). Also, defines the functions for writing log entries.

mm-pnp command line

*Run **mm-pnp** to demonstrate how to access, extract, and play mediastore content*

Synopsis:

```
mm-pnp [-c config_file] [-v[v...]]
```

Options:

-c *config_file*

Specify a configuration file that defines synchronization and playback policies for specific device types.

If you don't provide the path for a configuration file, **mm-pnp** reads the default file (**/etc/mm/mm-pnp.conf**).

-v

Increase output verbosity. Messages are written to **sloginfo**.

The `-v` option is handy when you're trying to understand the operation of **mm-pnp**, but when lots of `-v` arguments are used, the logging becomes quite significant and can change timing noticeably. The verbosity setting is good for systems under development but should probably not be used in production systems or during performance testing.

Description:

The **mm-pnp** program demonstrates the process of detecting a new mediastore, uploading its media information to a database, and playing its media files.

The program constantly monitors the PPS mount objects to readily detect when the user attaches a mediastore. In response to this user action, **mm-pnp** invokes the multimedia services necessary to synchronize and play the mediastore's content.

You can modify the *configuration file* (p. 38) to customize how **mm-pnp** uses those other services to manage content read from a particular device type. These settings tell **mm-pnp** where to look in the mediastore filesystem to find media files, whether or not to automatically start playback, and more.

Once started, **mm-pnp** runs as a self-contained process that doesn't require any user input or accept any commands. The program remains active, responding as configured when the user attaches mediastores, until you forcibly terminate the program with **Ctrl–C** or the *kill()* command.

Configuring mm-pnp

You can configure **mm-pnp** by changing the settings in its configuration file. These settings define the directories and device paths used for writing media databases and managing playback. They also define parameters that control how media content from specific device types is synchronized and stored.

The command line for starting **mm-pnp** doesn't support any configuration options; you must use the configuration file to define any nondefault settings for multimedia services and to customize media synchronization and database management for different device types. The configuration file offers a convenient way to change the behavior of **mm-pnp** without modifying and recompiling the code.

The **mm-pnp** program defines default values for all configuration options so there are no mandatory option settings for the configuration file.

Configuration file contents

The configuration file is a text file with one setting defined per line. Each line consists of an option name, followed by the equal sign (=), followed by an option value. For readability, you can enter comments by starting lines with the number sign (#).

The first section of the file defines settings that affect the operations of the other multimedia services used by **mm-pnp**. These settings are independent of the device type, and they include:

- the hardware device that **mm-renderer** uses for audio output
- the PPS directories to scan for mount objects and QDB database configuration objects
- the mountpoints used by each multimedia service (expressed as device paths in **/dev**)
- the prefix that **QDB** uses in database names
- the maximum number of concurrent synchronizations across all device types

In the default configuration file (**/etc/mm/mm-pnp.conf**), the first section looks like this:

```
audio_device=snd:/dev/snd/pcmPreferredp
qdb_mountpoint=/dev/qdb
renderer_mountpoint=NULL
pps_pub_root=/pps/qnx
pps_qdb_root=/pps/qnx
qdb_db_modifier=db_
sync_mountpoint=/dev/mmsync
sync_max=5
```

The subsequent sections define settings that affect how **mm-pnp** uses the other multimedia services to upload and store media information read from particular device types. These settings include:

- the directory used to hold the raw storage files for media databases
- the files that define the schema and initial contents of databases
- a flag indicating whether to play a device's media content right after synchronizing that content
- the location within the device's filesystem to look for media files
- other flags that control which information gets synchronized

Each of these sections must begin with a line that names the device type whose media operations are being configured (with the name enclosed in square brackets). The name of the device type must be one of the following:

- `device_unknown`
- `device_local`
- `device_usb`
- `device_ipod`
- `device_pfs`
- `device_audiocd`
- `device_datacd`
- `device_mmc`
- `device_sd`

In the default configuration file, the section that configures USB devices looks like this:

```
[device_usb]
sync_max=1
sync_path=/
sync_mask=0x4003
play_device=true
db_directory=Filename::/fs/tmpfs/
db_schema=SchemaFile::/etc/mm/sql/mmsync.sql
db_data_schema=DataSchemaFile::/etc/mm/sql/mmsync_data.sql
```

Global vs. device type synchronization limits

The global synchronization limit defined in the first section of the file applies over all type-specific synchronization limits defined in subsequent sections. For example, suppose you set `sync_max` to `5` in the first section but also set `sync_max` to `3` in the `[device_usb]` section. If **mm-pnp** starts two synchronizations for USB devices while three synchronizations for other device types are in progress, the program won't start a third synchronization for a USB device. No more content from USB devices can be uploaded or played until one of the current synchronizations completes.

You can set `sync_max` to `0` to disable media synchronization for a particular device type or for all device types.

Initialization and termination activities

Before it can synchronize and play media content, **mm-pnp** must parse its configuration file, configure other services, and start child threads. When the main thread receives the termination signal, it stops the child threads and cleans up resources.

By understanding the initialization and termination activities of **mm-pnp**, you can modify the program to log different information or support new configuration options. Or, you can use the code as a reference for writing multithreaded media apps.

The main thread of **mm-pnp** performs the following tasks:

1. **Setting up logging**

   At startup, the main thread initializes the logging service and adjusts the verbosity based on the `-v` command-line options. Messages outputted to **sloginfo** by **mm-pnp** are tagged with "`mm-pnp`".

2. **Parsing the configuration file**

   The main thread parses the configuration file and stores the configuration option settings. The **mm-pnp** program uses separate symbol tables when parsing different file sections to remember which settings apply globally and which ones apply to individual device types (see "*Configuring mm-pnp* (p. 38)" for information on the file sections).

3. **Setting up multimedia services**

   To enable media synchronization and playback, **mm-pnp** connects to **QDB**, **mm-sync**, and **mm-renderer**. It also creates lists to hold the information needed by these services, which includes:

   - *mediastore profiles*, which store the mountpoints and device types of attached mediastores
   - details on pending and on active synchronizations
   - playlists ready to be played

4. **Launching child threads**

   After the program setup is complete, the main thread launches three child threads:

   Device monitoring thread

   Reads a PPS object to obtain mountpoint information on newly attached devices, and then passes this information to **QDB** to load media databases.

   Database monitoring thread

   Monitors QDB status objects to learn when databases finish loading. In response, the thread invokes **mm-sync** to start synchronizing the media information from the corresponding devices.

   Synchronization event-processing thread

   Reads **mm-sync** events to learn when media file information has been synchronized. In response, the thread creates playlists based on the synchronized information and invokes **mm-renderer** to play them.

   These child threads work with other multimedia services to automatically extract media information and play media files when the user attaches a device (for details, see "*Device monitoring,* *synchronization, and playback* (p. 41)”). After launching the child threads, the main thread begins monitoring signals to wait for the termination request.

5. **Shutting down**

   The **mm-pnp** programs runs until you issue the termination signal by using **Ctrl–C** or the *kill()* command. When you tell it to shut down, the main thread:

   - terminates the child threads
   - disconnects from the multimedia services
   - destroys the lists that store mediastore, synchronization, and playlist information

Device monitoring, synchronization, and playback

An **mm-pnp** child thread monitors a device-related PPS object and indirectly invokes **QDB** to load the appropriate database in response to a device attachment. Then, a second thread synchronizes the device's media content using **mm-sync**. Finally, a third thread plays the content using **mm-renderer**.

These tasks are performed in an automated, ongoing process. Understanding this process is essential if you want to adapt **mm-pnp** to demonstrate a different synchronization and playback policy, or if you want to use the code as a basis for your own media apps.

When the user attaches a device, the **mm-pnp** threads and the multimedia components work together to synchronize and play media content, as illustrated here:

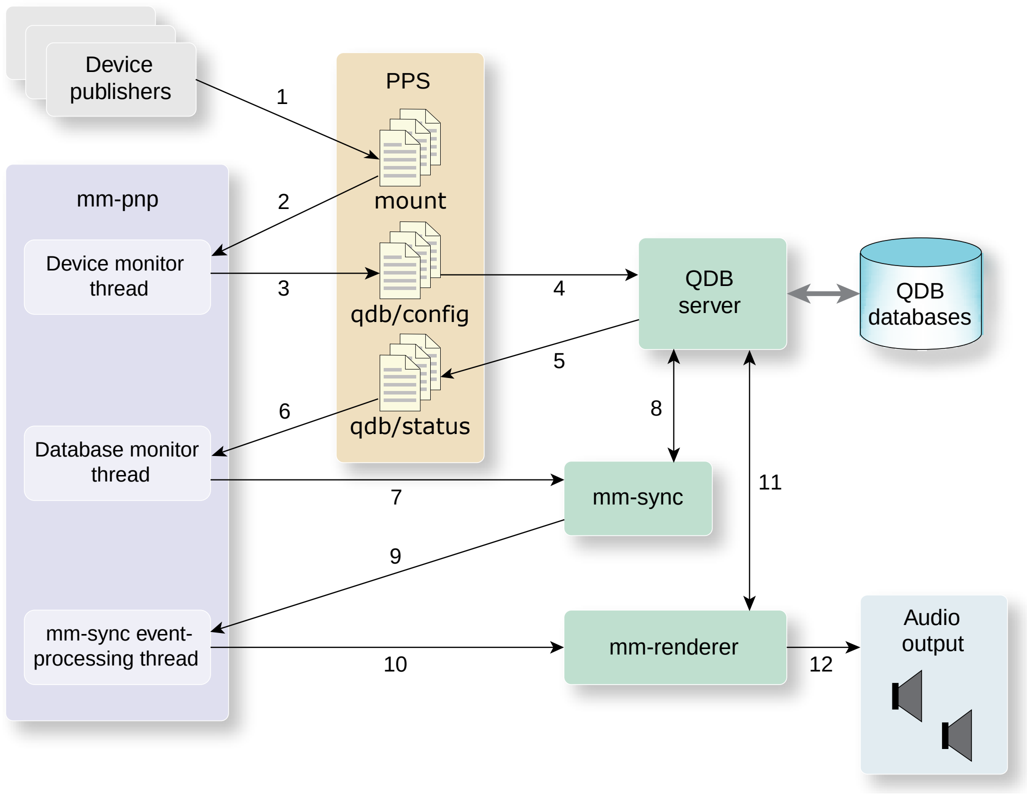

Figure 1: Interaction between mm-pnp and multimedia components

The synchronization and playback process is triggered when the user attaches a device. At that point, **mm-pnp** and the multimedia components perform the following tasks:

1. **Reporting device attachments**

   The device publisher that monitors the hardware events triggered by the device's attachment (e.g., USB bus notifications or SD card insertions) writes the device's information to PPS.

2. **Extracting device information**

   The **mm-pnp** *device monitoring thread* reads this information, which includes the mountpoint and device type, from PPS, then saves the information in the *mediastore profiles list*.

3. **Publishing the database configuration**

   Based on the database settings defined for the device type, the same thread publishes the configuration for the QDB database that will store the device's media information.

4. **Loading the database**

   When **mm-pnp** publishes the PPS object to the QDB configuration directory (**/pps/qnx/qdb/config/**), the QDB server begins loading the device's database.

5. **Reporting database status**

   The QDB server reports the outcome of the database load operation by writing the database's new status to a PPS object in the QDB status directory (**/pps/qnx/qdb/status/**).

6. **Looking up device information**

   The *database monitoring thread* reads database statuses from a special PPS object (by default, **/pps/qnx/qdb/status/.all**). When it learns that a database has been loaded successfully, the thread uses the database name to look up the corresponding device's information in the mediastore profiles list. From this list, the thread obtains the device's mountpoint and synchronization path, which it then stores in a new entry in the *pending synchronizations list*.

7. **Starting synchronizations**

   Before it can start a synchronization, **mm-pnp** must check the limits on the number of concurrent synchronizations allowed for all device types and for the device type of the first entry in the pending synchronizations list. If these limits have *not* been exceeded, the database monitoring thread passes the mountpoint and synchronization path stored in the first list entry to **mm-sync**, which starts synchronizing the device's media information. Then, the thread moves the entry from the pending synchronizations list into the *active synchronizations list*.

   If the synchronization limits have been exceeded, **mm-pnp** instead waits for one of the active synchronizations to finish and when this happens, it starts another synchronization.

   If you extend the `DB manager` (p. 36) module to define additional operations for handling database status changes, the database monitoring thread should perform these operations right after it calls **mm-sync**.

8. **Writing file information to the database**

   The **mm-sync** service reads file information from media tracks on the device, then invokes **QDB** to store this information in the device's database (which was loaded in Step *4* (p. 42)).

9. **Notifying about file information uploading**

   When **mm-sync** finishes uploading the file information to the database, the service generates the `MMSYNC_EVENT_MS_1PASSCOMPLETE` event. The *synchronization event-processing thread* receives this event and in response, looks up the device's entry in the active synchronizations list to obtain the database name.

10. **Specifying a playlist**

    The same thread uses the database name to define a playlist for playing the device's media content. The playlist is based on an SQL query that retrieves information from all the files listed in the database.

    The thread then invokes **mm-renderer** to:

    a. create a new context
    b. attach the output to the default sound device to enable audio output
    c. attach the newly defined playlist as the input
    d. start playing the playlist

    Because you can name only one output device for **mm-renderer** to use, **mm-pnp** stops any current playback when it processes an `MMSYNC_EVENT_MS_1PASSCOMPLETE` event. So, the program always begins playing the media content of the latest device attached to the system.

    ---

11. **Querying filenames in the database**

    The **mm-renderer** service extracts the names of all media files listed in the database by running the SQL query that **mm-pnp** provided as the playlist basis. The service must know the filenames to play their media content.

12. **Outputting playlist tracks**

    The **mm-renderer** service then starts playing the tracks in the playlist. The tracks are played in sequence, with no repeating, until the entire playlist has been played.

Index