Multimedia Player Developer's 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: Thursday, April 17, 2014

Table of Contents

About This Guide

The *Multimedia Player Developer's Guide* is aimed at CAR programmers who want to write applications that use the `mm-player` service to browse mediastore content and play tracks.

This table may help you find what you need in this guide:

To find out about:	Go to:
The capabilities of the `mm-player` service	*Multimedia Player Overview* (p. 9)
The plugins used by `mm-player` to interface with different devices	*Media Player Plugins* (p. 17)
How to change the command options given to `mm-player` by SLM during bootup	*Running mm-player* (p. 25)
The command line for starting `mm-player` manually	*Command line for mm-player* (p. 34)
Connecting to `mm-player` and browsing and playing media content through its API	*Multimedia Player API* (p. 37)

Typographical conventions

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

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl** –**Alt** –**Delete**
Keyboard input	`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.

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

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

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
Multimedia Player Overview

The multimedia player service, `mm-player`, allows applications to browse and play mediastore content. Through `mm-player`, applications can obtain a list of available media sources, explore the filesytems and read media metadata from those sources, and play tracks.

The `mm-player` service is a media browsing and playback engine that uses lower-level system components to help it execute media commands sent by client applications. The system components allow `mm-player` to navigate the filesystems of mediastores, read metadata from media files, and manage media flows during playback.

The `mm-player` service can access many types of media sources:

- local drives
- USB storage devices
- iPods
- DLNA devices
- devices paired with Bluetooth
- files on MTP devices

Starting with this release, `mm-player` replaces `mm-control`. Unlike its predecessor, `mm-player` can browse mediastore content (as well as play it). Also, `mm-player` offers a simplified set of playback commands so your media applications don't have to create and attach output zones or define track and input parameters.

Architecture

Media Player and other applications can send requests to `mm-player` to browse and play media. The `mm-player` service uses plugins to explore filesystems, retrieve media metadata, and manage playback on different device types.

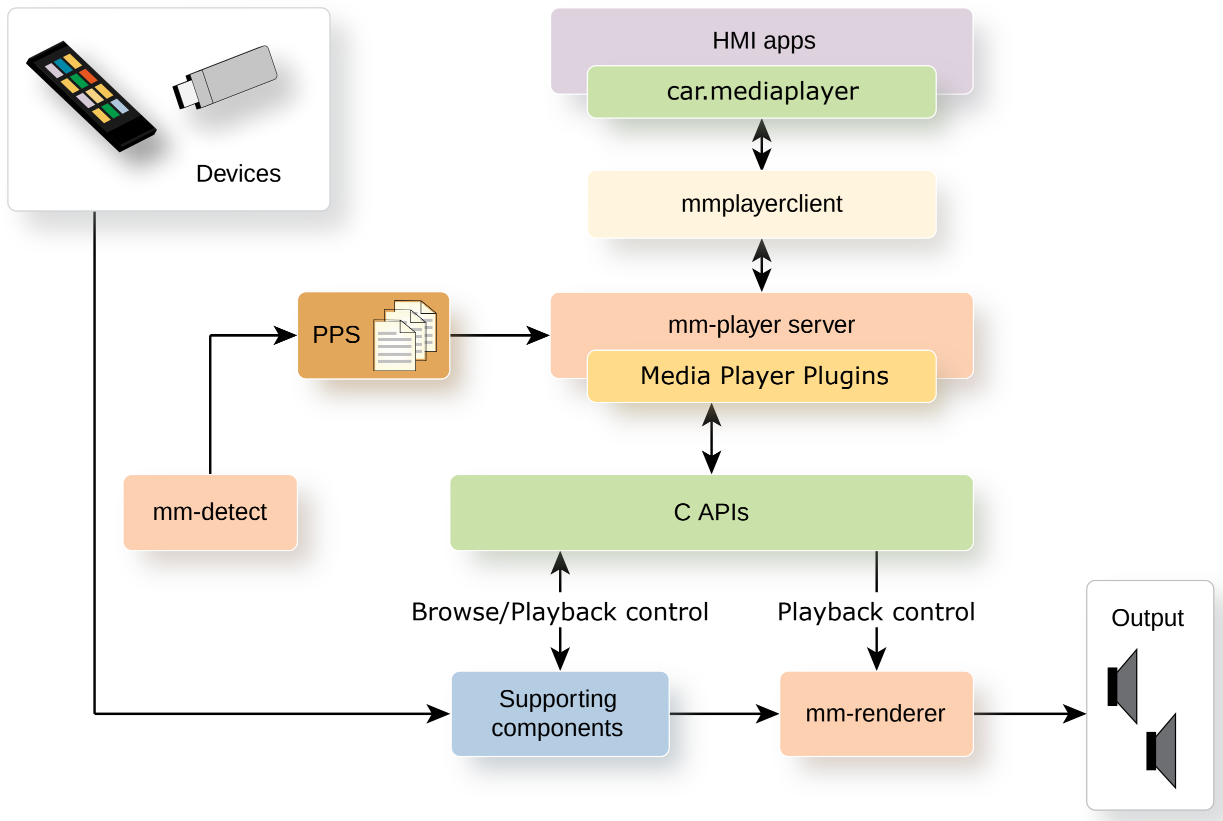

Figure 1: Architecture of mm-player

Media Player and other HMI apps can use a WebWorks extension (`car.mediaplayer`) to talk to `mm-player`. This extension translates JavaScript function calls into calls to the C functions in the `mm-player` client library (`mmplayerclient`). You can write other HTML5 applications that use this same extension to send media requests to `mm-player`. Or, you can write applications in C that directly call the `mm-player` API.

The client library forwards the function calls to the `mm-player` server, which implements the media browsing and playback engine. To learn which media devices are accessible for browsing and playback, the `mm-player` server reads the status object published through PPS by the multimedia detector service, `mm-detect`. This status object (`/pps/services/mm-detect/status`) lists the devices currently attached to the system. The `mm-player` server subscribes to this object and receives updates whenever its contents change, which happens when a device is added or removed. This way, `mm-player` always has an up-to-date list of available media sources.

When it receives a request to browse or play content on a particular media source, `mm-player` selects the Media Player Plugin (MPP) designed for the media source's device type (e.g., iPod, Bluetooth, DLNA). Plugins abstract the details of communicating with media sources and make `mm-player` extensible for supporting new device types (see "*Media Player Plugins* (p. 17)" for more information).

MPPs use lower-level components, such as device drivers and other OS services, to browse the filesystems of media sources, retrieve track metadata, and process playback commands. For playback, MPPs use both the supporting components and the `mm-renderer` service to manage media streams between media sources and output hardware (e.g., speakers).

To perform any playback operation, `mm-player` must have an active track sequence (or *tracksession*), which tells it which media file to play next. MPPs manage their own tracksessions.

In this release, `mm-player` doesn't support video playback (only audio). To play videos, your applications must use the HTML5 video features.

Layers of mm-player

The `mm-player` service uses a layered design to separate the application logic for the tasks of managing client connections, validating commands, maintaining media source information, managing tracksessions, and interfacing with devices.

The `mm-player` service consists of three layers:

Client layer

This layer implements the client-side `mm-player` library (`libmmplayer`), which:

- creates and returns connection handles to the client application
- detects basic parameter errors such as empty strings used for player names or media source IDs
- forwards API commands and their parameters to the `mm-player` server

Server layer

This layer implements the `mm-player` server, which:

- opens and closes connections to players
- validates parameters before forwarding commands to the plugin layer

- publishes updates to player states through PPS
- manages tracksessions
- maintains a list of media sources

Plugin layer

This layer consists of many plugins, each of which:

- notifies the `mm-player` server of important events, including but not limited to:
  - a media source connection or disconnection
  - a user-requested change in the playback position
  - an error occurred when browsing or playing media
- creates and returns handles to "play sessions" and "browse sessions" to the server layer, allowing that layer to direct commands to specific media sources
- uses platform services to carry out playback commands (e.g., `play`, `stop`, `seek`)

Players

Using `mm-player`, media applications can create and configure *players* to browse media sources, retrieve track metadata, and control playback. All media operations in `mm-player` must be applied to a particular player.

Each player operates independently of the others, so the browsing and playback commands that you send it affect only its own tracksession and media playback. You can assign to players whatever names you want to use to distinguish them, and you can create as many players as you need. Using multiple players lets you direct media output to different zones (i.e., groups of output devices). For instance, you can use separate players for the front and rear speakers of a vehicle.

Before you can browse or play media content, you must create at least one player so you have somewhere to direct your media commands. When it creates a player, the `mm-player` service returns a player handle, which you must provide in subsequent commands to indicate which player the command is applied to. When you're finished using a player, you must tell `mm-player` to destroy it.

Media sources

Media sources are `mm-player` objects that represent devices that store media content (*mediastores*). Applications can request a list of available media sources from `mm-player` and then select a specific media source to browse or to play media files.

Each media source object contains a media source ID and name field (to distinguish it from other media sources), as well as device information such as the unique ID (UID), hardware type, and supported playback operations.

Clients must pass in a media source ID to `mm-player` when browsing the contents of mediastores and when retrieving metadata from media files. Also, each tracksession (i.e., track sequence) is associated with a media source, so clients must also provide a media source ID when creating tracksessions.

Media nodes

Media nodes represent entries in the filesystems of media sources. Applications can obtain, from `mm-player`, the list of media nodes found in a certain path on a media source and then select a particular media node to browse more specific paths or to define tracksessions.

Each media node may correspond to an audio, video, or photo file or a folder. The file type is stored in the media node object, which also stores the media node ID and name (to distinguish it from other media nodes), information on the media source that stores the media node, and the number of children. For folders, when this last field is greater than `0`, this means they contain other media nodes, which could be subfolders or media files. Applications must check each folder's number of children and retrieve the media nodes in any subfolders to explore content at all directory levels.

Clients must pass in a media node ID to `mm-player` when browsing a media source's contents, retrieving media metadata, and creating tracksessions (i.e., track sequences). To start browsing the contents of a new media source whose directory structure is unknown, clients must specify the root directory (`/`) as the media node ID.

Tracksessions

Tracksessions represent playback sequences of tracks (media files). To perform any playback operation, a player must have an active tracksession so it knows which track is currently selected for playback and the relative order of the tracks.

A player must monitor the current track selection so it knows which media file the user wants to play and so it can provide them with information about any media content being played or scheduled to be played next. Knowing the playback order is necessary to update the current track information when carrying out operations such as *next* or *previous*.

Each tracksession object stores only a tracksession ID (to distinguish it from other tracksessions) and the number of tracks in the playback sequence. In this release, this object doesn't need to reference the individual tracks because players support only one tracksession at a time. This design could change in future releases to support multiple tracksessions per player.

With the current design, when a client creates a tracksession based on a media node ID, the tracksession becomes the only active one for the player. Clients must then pass this ID in when retrieving either the tracks in the tracksession or the metadata of the current track, and when selecting a new track to play. The tracksession object as well as the index (playback position), media node object, and metadata of the current track are stored in the player's state information, which clients can obtain through `mm-player` commands.

Chapter 2
Media Player Plugins

The `mm-player` service uses Media Player Plugins (MPPs) to support browsing and playing media from different device types.

Each MPP interfaces with a specific type of media device. This modular design isolates device functionality from the common browsing and playback functionality of `mm-player`. The MPPs communicate with system components through C APIs to navigate mediastore filesystems and read metadata is done through C APIs. The sections that follow explain how these plugins and their supporting components fit into the overall `mm-player` and CAR multimedia architecture.

POSIX plugin

The `mm-player` service uses the POSIX plugin to browse and play media files stored on devices with POSIX filesystems. Such devices include USB sticks, SD cards, or CDs.

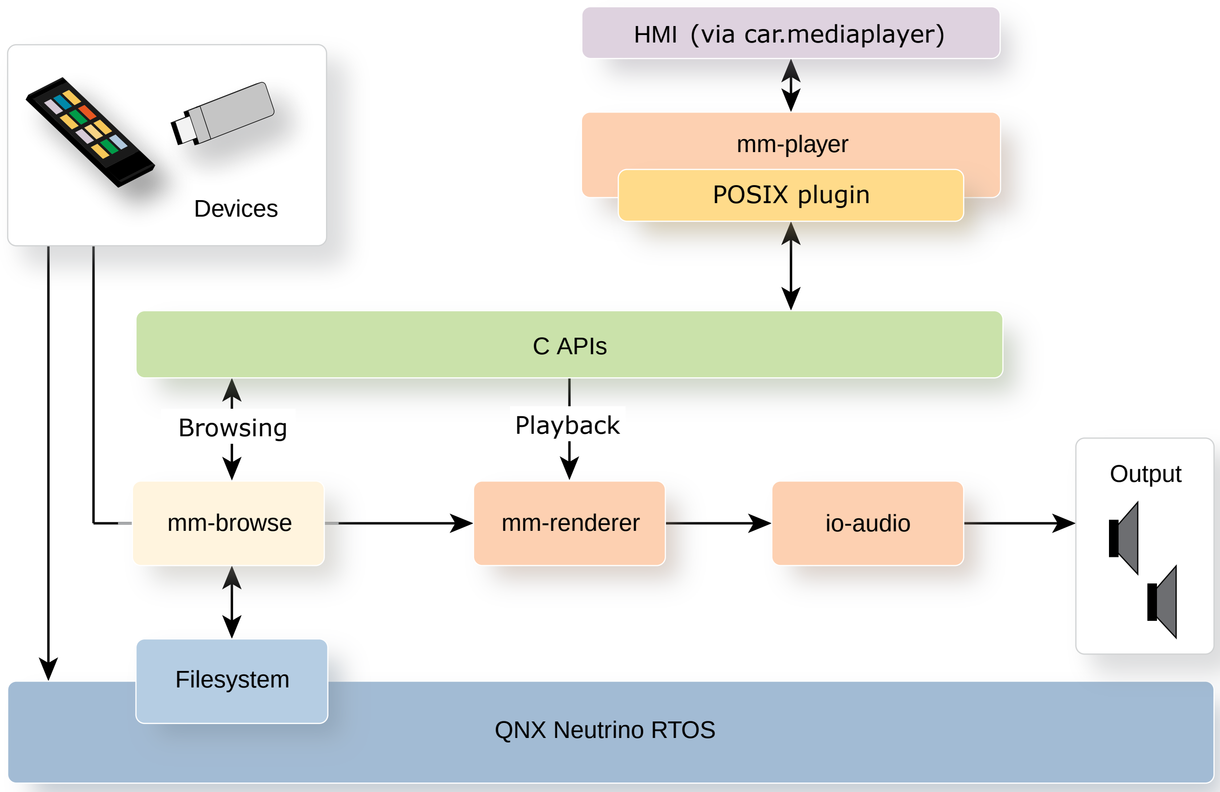

Figure 2: POSIX Plugin Architecture

The POSIX plugin uses C APIs to communicate with the `mm-browse` library for browsing media files and with the `mm-renderer` service for managing playback. When devices with POSIX filesystems are attached to the CAR system, services in the QNX OS layer mount the device filesystems. The plugin uses the `mm-browse` library to search these mounted filesystems and read metadata from their files. To deliver the metadata to the HMI, `mm-player` uses the `car.mediaplayer` WebWorks extension (for more information, see the multimedia architecture overview in the *QNX CAR Multimedia Architecture Guide*).

When the user starts playing a media file stored on a POSIX device, the POSIX plugin passes the appropriate URL to `mm-renderer` as the input. The `mm-renderer` service reads the input media off the device and then sends the media content to `io-audio` for output.

AVRCP Plugin

The `mm-player` service uses the AVRCP (Audio/Video Remote Control Profile) plugin to browse and play media files stored on devices accessed over a Bluetooth connection.

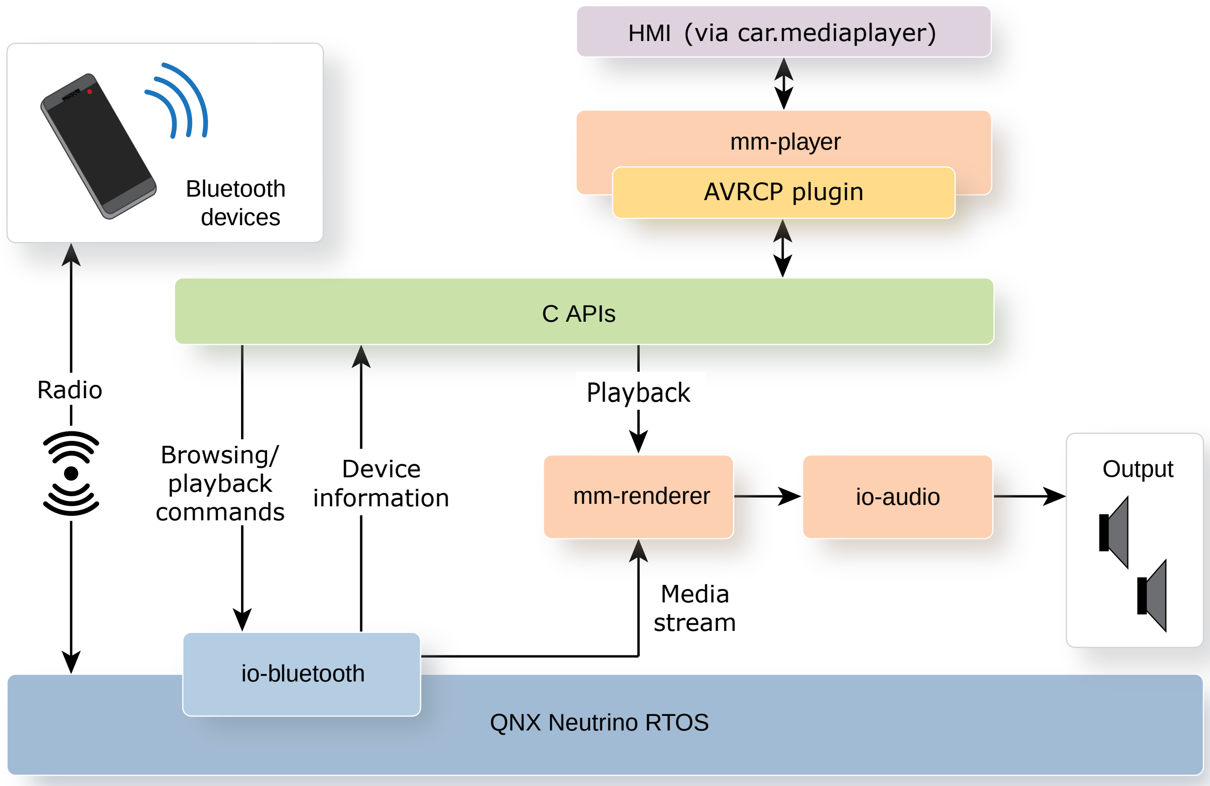

Figure 3: AVRCP Plugin Architecture

The AVRCP plugin communicates with the `io-bluetooth` service (through standard file I/O calls) to browse media content on Bluetooth-connected devices. Media information from Bluetooth devices arrives over a radio connection. A Bluetooth device driver (not shown in *Figure 3: AVRCP Plugin Architecture* (p. 19)) passes this information to `io-bluetooth`. The plugin can then retrieve all device information—position data, connectivity state, and media file metadata—through `io-bluetooth`. To send this information to the HMI, `mm-player` uses the `car.mediaplayer` WebWorks extension (for more information, see the multimedia architecture overview in the *QNX CAR Multimedia Architecture Guide*).

When the user starts playing a media file accessible through Bluetooth, the AVRCP plugin sends playback commands to `io-bluetooth` and `mm-renderer` through their C APIs. The `io-bluetooth` service initiates the media stream on the Bluetooth-connected device and directs that stream to `mm-renderer` as input. The `mm-renderer` service then sends the stream to `io-audio` for output.

DLNA Plugin

The `mm-player` service uses the DLNA (Digital Living Network Alliance) plugin to browse and play media files stored on devices compliant with the Universal Plug and Play (UPnP) standard.

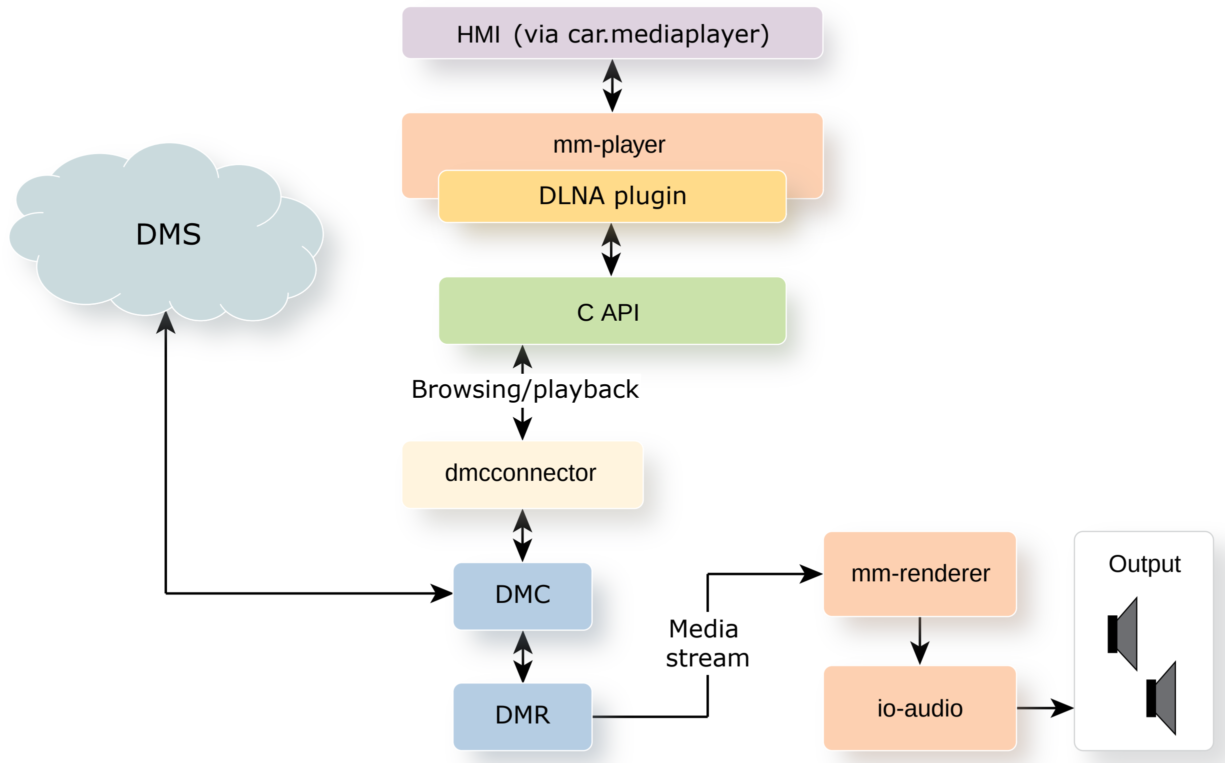

Figure 4: DLNA Plugin Architecture

In this CAR release, the supported UPnP device type is a Digital Media Server (DMS) accessed over the network. The DLNA plugin uses a C API to communicate with the `dmcconnector` library for browsing and controlling playback of media files managed by a DMS. The `dmcconnector` library handles communication between the Digital Media Controller (DMC), the Digital Media Renderer (DMR), and the DMS. Through its browsing operations, the library allows the plugin to retrieve metadata on media files. To provide this metadata to the HMI, `mm-player` uses the `car.mediaplayer` WebWorks extension (for more information, see the multimedia architecture overview in the *QNX CAR Multimedia Architecture Guide*).

When the user starts playing a media file managed by an accessible DMS, the DLNA plugin sends playback commands to `dmcconnector`. The active DMR processes the media stream and passes it as input to `mm-renderer`, which then directs it to `io-audio` for output.

iPod Plugin

The `mm-player` service uses the iPod plugin to play media files stored on iPods.

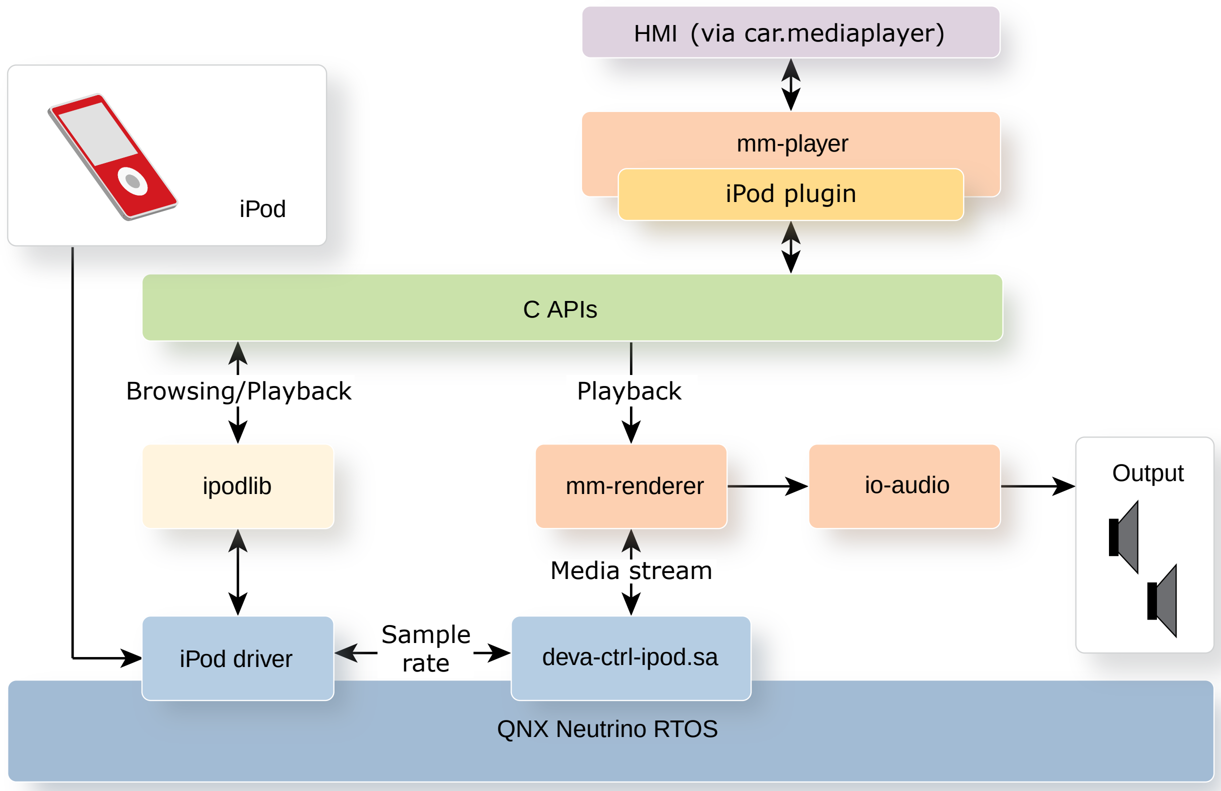

Figure 5: iPod Plugin Architecture

Note that this plugin doesn't support the `mm-player` browsing operations. This means that in the HMI, you can see iPod media content only after it's been synchronized to QDB databases. For information on synchronization, see the "*synceddb Plugin* (p. 23)" section.

The iPod plugin uses C APIs to communicate with the `ipodlib` library for internally browsing and managing playback of iPod media content, and with the `mm-renderer` service for managing playback.

When the user plays media files stored on iPods, the plugin directs playback commands to both `ipodlib` and `mm-renderer`. The `ipodlib` library talks to the iPod driver, which sends sample-rate updates to the active audio capture driver (e.g., `deva-ctrl-ipod.sa`) to control the media stream. This driver receives the media stream from the iPod device, processes the stream, and passes it as input to `mm-renderer`. The `mm-renderer` service then directs the media stream to `io-audio` for output.

To provide playback status updates to the HMI, `mm-player` uses the `car.mediaplayer` WebWorks extension (for more information, see the multimedia architecture overview in the *QNX CAR Multimedia Architecture Guide*).

synceddb Plugin

The `mm-player` service uses the synceddb plugin to browse media files that have had their metadata synchronized to QDB databases.

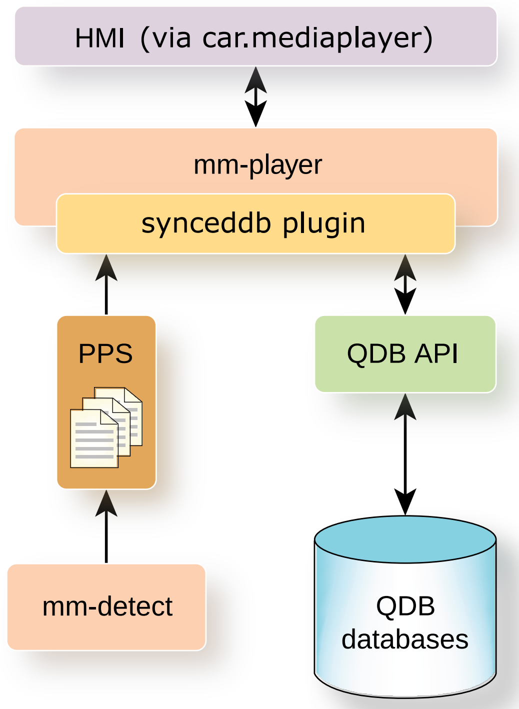

Figure 6: synceddb Plugin Architecture

When the user attaches a device, the `mm-detect` service loads the QDB database that stores the device's media metadata and then publishes the device's information to a special PPS object (/pps/services/mm-detect/status). When the user removes the device, `mm-detect` unloads its database and updates the same PPS object to indicate that the device is no longer present. The synceddb plugin monitors this object and updates its list of accessible devices when the object's contents change. This design means that the plugin can browse the contents of only those mediastores currently attached to the system.

The synceddb plugin allows `mm-player` to quickly provide the HMI with media information about frequently used devices. For example, suppose the user attaches a new iPod that contains a lot of media content. Because the *iPod Plugin* (p. 21) doesn't support browsing, the user can't see information about the iPod's media files until their metadata has been synchronized (at which point it can be retrieved from the populated QDB database). If the same iPod is detached and then reattached, its database will be reloaded and the synceddb plugin can immediately retrieve the metadata and pass it to the HMI.

Because it reads information from QDB databases and not external devices, the synceddb plugin doesn't need any supporting components to retrieve metadata. The QDB API is implemented by a C library that the plugin links into its codebase, so the plugin doesn't talk to another process but instead simply queries the databases and stores the results. To send metadata to the HMI, `mm-player` uses the `car.mediaplayer` WebWorks extension (for more information, see the multimedia architecture overview in the *QNX CAR Multimedia Architecture Guide*).

Chapter 3
Running mm-player

Client applications don't have to start `mm-player` in the same way that they must run other multimedia services at specific times to perform tasks. CAR systems use the System Launch Monitor (SLM) service to start `mm-player` during bootup. Applications should manually start `mm-player` only for recovery purposes.

Starting mm-player with command options during bootup

SLM automates process management by starting processes in an order that respects their interprocess dependencies. The list of processes to launch and their properties, including their command-line arguments and interprocess dependencies, is written in a configuration file (`/etc/slm-config-all.xml`). For the full explanation of how SLM works, see the “System Launch and Monitor (SLM) section” of the *System Services Reference*.

Using SLM to start `mm-player` ensures that the in-car system can browse and play media as soon as the system finishes booting up and also that `mm-player` runs with the same command options and therefore behaves consistently from one bootup to the next.

SLM is preconfigured to start `mm-player` with specific command options but you can change these options to better suit the needs of your in-car system.

To change the command options passed by SLM to `mm-player`*:*

1. From a command console connected to your in-car system, navigate to and open the SLM configuration file, whose full path is `/etc/slm-config-all.xml`.
2. In the configuration file, locate the component that specifies the properties for `mm-player`.

   This component is the `<SLM:component>` XML object with the name `"mmplayer"`.
3. Change the value of the `<SLM:args>` tag in the `"mmplayer"` component to hold the new set of `mm-player` command-line options.

   For the full list of available command-line options, see “*Command line for mm-player* (p. 34)”.
4. Save the changes to the SLM configuration file and return to the console.
5. If you want the new configuration to take effect immediately, in the console, enter `reboot`.

The system reboots and the SLM utility relaunches all processes, including `mm-player`, with their command options given in the configuration file. When the system finishes reloading, `mm-player` is running with the new configuration.

If you don't reboot immediately after changing the configuration file, `mm-player` continues to run with its previous configuration, until you shut down the system and restart.

Restarting mm-player

An application can restart `mm-player` with an explicit command if the service has stopped running and the application can't proceed without support for media browsing and playback.

A client application should restart `mm-player` manually only if:

- the `mm-player` process has terminated unexpectedly
- the client is actively browsing or playing media
- the client has enough system configuration knowledge to set the appropriate command-line options for `mm-player`
- it is undesirable to reboot the in-car system to restart `mm-player` because doing so would be too disruptive to the user

To restart `mm-player` manually, an application must:

1. Confirm that `mm-player` isn't already running by checking the list of active processes with the `pidin` or `ps` command.
2. Confirm that the `nowplaying`, `mm-detect`, and `mm-renderer` processes are already running by checking the list of active processes.

   The `mm-player` process depends on these three other services, so if any one is *not* running, that service must be started; otherwise, `mm-player` won't work properly.
3. Send the `mm-player` command line containing the desired options to the OS, using the *system()* or *spawn()* system call.

   The OS tries to run `mm-player` and reports the outcome to `sloginfo`.

   For details on the system calls that send commands to the OS, see the *C Library Reference*. For the full list of command-line options, see "*Command line for mm-player* (p. 34)".

If `sloginfo` shows no error, `mm-player` is now running.

Your system should run only one instance of `mm-player`, so applications relying on the service must coordinate with each other to avoid starting `mm-player` multiple times.

SLM specification of mm-player

The SLM configuration file stores the list of processes to be automatically launched and their interprocess dependencies. In CAR systems, this list includes `mm-player` and its prerequisite and dependant programs. You can change the configuration file to run `mm-player` with different settings.

The following excerpt from the SLM configuration file shows some (but not all) of the property settings for `mm-player`:

```
<SLM:component name="mmplayer">
    <SLM:command>mm-player</SLM:command>
    <SLM:args>-c /etc/mm-player.cfg</SLM:args>
    <SLM:depend>nowplaying</SLM:depend>
    <SLM:depend>mmtrkmgr</SLM:depend>
    <SLM:depend>mmdetect</SLM:depend>
    <SLM:depend>mmrenderer</SLM:depend>
</SLM:component>
```

Here, the name `mmplayer` assigned to the `<SLM:component>` XML object is an internal label used within the configuration file. This label differs from the process name of `mm-player`, which is provided in the `<SLM:command>` tag.

Command-line arguments

The `<SLM:args>` tag lists the command-line arguments. By default, only the path of the `mm-player` configuration file is specified (with the `-c` option), but you can change the value assigned to `<SLM:args>` to include other options. The new `mm-player` settings will take effect after you reboot the car system, causing SLM to relaunch the service. See "*Running mm-player* (p. 25)" for instructions on changing the command settings for `mm-player`.

Workflow

The `<SLM:depend>` objects list which processes must be running before `mm-player` can be started. These dependencies mean that SLM must ensure that the `nowplaying`, `mm-detect`, and `mm-renderer` processes are running before starting `mm-player`.

The programs that are prerequisites to `mm-player` have their own prerequisites. For instance, `mm-renderer` requires the Audio Manager service (represented in SLM by the `audioman` object) to be running so that the audio components of the media that gets played can be output over hardware. The Audio Manager and many other services depend on PPS. The interprocess dependencies between `mm-player` and the services that it uses make up a complex workflow of processes. The following illustration shows part (but not all) of this workflow:

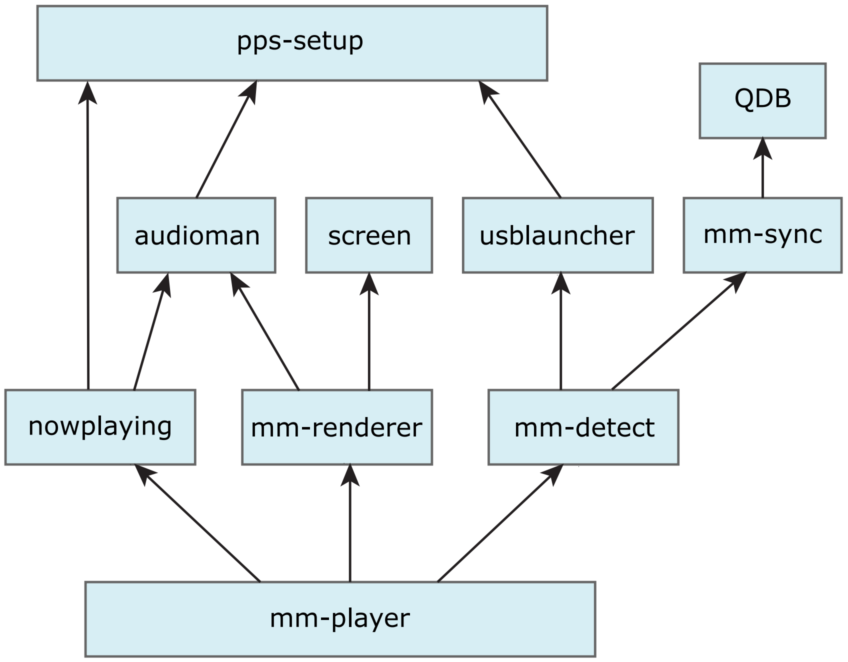

Figure 7: Workflow of mm-player and related processes

Configuration file

The `mm-player` configuration file defines playback settings, such as the audio output device and the progression mode for randomized playback. The file also defines settings for individual Media Player Plugins (MPPs), such as whether they manage their own tracksessions.

The QNX CAR Platform for Infotainment includes a default `mm-player` configuration file (`/etc/mm-player.cfg`). You can modify this included file or create your own. To use your own file, you must specify its path in the `-c` option on the `mm-player` command line.

The configuration file must be in JSON format. Two top-level JSON objects can be defined: a `“player”` object, which lists playback settings to assign to players, and a `“plugins”` object, which lists properties of the various plugins (MPPs). In either object, member fields must consist of key-value pairs, with both elements encoded as strings.

The player object

The `player` object can contain the following fields:

audio_output

The URL naming the audio output device. For details on the required URL format and its meaning, see “mmr_output_attach()” in the *Multimedia Renderer Developer's Guide*.

video_output

The URL naming the video output device. For details on the required URL format and its meaning, see “mmr_output_attach()”.

Although `mm-player` presently doesn't support video playback, this field is required for certain plugins to work. In this release, you can use the URL value given in the default file.

update_interval

The frequency (in milliseconds) of the track position updates sent by MPPs to the `mm-player` server. The server publishes these position updates in PPS.

The default frequency is 750 ms.

progression_mode

The method `mm-player` uses to fetch the next track during randomized playback. Can be one of two values: `“rosp”` (random order sequential progression) or `“sorp”` (sequential order random progression).

The default setting is `“sorp”`.

lastmode_audio

This nested object configures the "last mode" audio restoration feature, in which the media player tries to resume playback following a system reboot that interrupted the previous playback. The `lastmode_audio` object contains these fields:

timeout

A time limit (in seconds) on how long the audio restoration feature should attempt to resume playback.

The default limit is `5` seconds.

max_try

The maximum number of tracks that the media player should look for when trying to resume playback. This setting supports the case where the previous track is unavailable because, for example, the user removed the media storing this track during the reboot. The media player first tries to resume playing the previous track (i.e., the one playing when the system rebooted) and then tries to play a fixed number of other tracks, based on the `max_try` setting.

The default setting is `5`.

device_type

The hardware type of the media accessed by this player. Can be one of `HDD`, `USB`, or `IPOD`.

The default setting is `HDD`.

recursion_depth

The number of directory levels to search to find tracks when building a tracksession from a "base" media node. When this value is set to `-1`, there's no depth limit on the directories searched. Otherwise, this value must be greater than `0`. When it's set to `1`, only the immediate directory of the base media node is searched.

The default depth is `4`.

To play media on POSIX devices, Bluetooth devices, or iPods, you must include the `audio_output`, `video_output`, and `update_interval` fields. The `progression_mode` field is optional.

The plugins object

The `plugins` object contains other objects that configure specific MPPs. Each of these objects must have the same name as the library file that implements the corresponding plugin. For example, the object for the POSIX plugin must be named “`mpp-default.so`”. The plugin objects can contain the following fields:

mode

The *playback mode*, which determines which layer manages the tracksessions. Can be one of two values: “`player`” (to indicate that `mm-player` manages the tracksessions) or “`device`” (to indicate that the plugin manages its own tracksessions). The default setting for all plugins is “`player`”.

view_name

Name of the view used for accessing metadata on devices supported by this plugin. The view affects how media information can be presented in the HMI. For example, the view may determine which extended metadata fields can be read from the device. The default configuration file uses the `synced` and `live` views, but you can define others to suit your HMI's needs.

audio_ext

A JSON-formatted string listing all the audio file extensions that the plugin recognizes. The plugin can browse and play only those audio files with extensions contained in this list.

You must escape the double quotes enclosing the individual list values with a backslash (\).

video_ext

A JSON-formatted string listing all the video file extensions that the plugin recognizes. The plugin can browse and play only those video files with extensions contained in this list.

You must escape the double quotes enclosing the individual list values with a backslash (\).

cfgfilename

Applies only to the synceddb plugin

The name of the configuration file. This field is mandatory for the "`mpp-synced_default.so`" object.

Default configuration file

The contents of the default configuration file look like this:

```
{
    "player":{
        "audio_output":"audio:default",
        "video_output":"screen:?dstx=320&dsty=85&zorder=100&dstw=460&dsth=259",
        "update_interval":"750",
        "progression_mode":"sorp",
        "lastmode_audio":{
            "timeout":"5",
            "max_try":"5",
            "device_type":["HDD","USB"]
        },
        "recursion_depth":"4"
    },
    "plugins":{
        "mpp-default.so":{
            "mode":"player",
            "view_name":"LIVE",
            "audio_ext":"[\"aac\",\"cda\",\"m4a\",\"m4b\",\"mp3\",\"wav\"]",
            "video_ext":"[\"m4v\",\"mp4\",\"mpeg4\",\"mov\",\"mpg\",\"mpeg\",\"3gp\",\"3g2\"]"
        },
        "mpp-synced_default.so":{
            "mode":"player",
            "view_name":"SYNCED",
            "cfgfilename":"anything"
        }
    }
}
```

Command line for mm-player

Start the multimedia player service

Synopsis:

```
mm-player [-b] [-c config_file] [-p dir] [-P priority]
    [-s service] [-U u[:g,...]] [-v]
```

Options:

-b

Run the `mm-player` process in the foreground (and not in the background). This option is handy for debugging, because it makes `mm-player` log messages to standard error in addition to `sloginfo`.

By default, `mm-player` runs in the background.

-c *config_file*

Specify an overridden path for the configuration file.

By default, `mm-player` uses the file at `/etc/mm-player.cfg`, but you can provide a path to any other valid configuration file.

-p *dir*

Specify a directory location for outputting the player state information (in PPS objects).

The default location is `/pps/services/mm-player/`.

-P *priority*

Set the priority of the `mm-player` process. When your system is busy running many applications, the priority level can have a considerable impact on the user experience when browsing and playing media.

The valid range is 1 to 63; the default is 15.

-s *service*

Specify the name of the `mm-player` server that receives media commands from the client library.

The server name must be in the form *name.provider*. The default name is `mm-player.sys`.

-U u[:g,...]

Run `mm-player` with the given user ID (`uid`) and possibly one or many group IDs (`gids`).

By default, `mm-player` inherits the caller's `uid` and `gids`, which means it runs as root because the service is started during bootup by SLM.

-v

Increase output verbosity. Messages are written to `sloginfo`. The `-v` option is cumulative, so you can add several v's to increase verbosity, up to seven levels.

Output verbosity is handy when you're trying to understand the operation of `mm-player`. However, 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 probably shouldn't be used in production systems or during performance testing.

Description:

The `mm-player` command starts the multimedia player service, which processes media browsing and playback commands sent through its C API. Client applications can invoke `mm-player` to obtain a list of available media sources, explore the filesytems and read media metadata from those media sources, and select tracks for playback.

Through command options, you can change the configuration filepath (to use a nondefault configuration) and the player state output directory (to output state information to another PPS directory). You can also assign a user ID, multiple group IDs, the priority level, and the debugging output level to the `mm-player` process.

Once started, `mm-player` can't adjust any of these settings. To reconfigure `mm-player`, you must restart it with a different command line. The command line is contained in the SLM configuration file, located at `/etc/slm-config-all.xml`.

Chapter 4
Multimedia Player API

The multimedia player API is the primary interface for accessing and controlling the `mm-player` service. Through this API, you can browse media sources, define tracksessions, and control playback. You can also receive API events to monitor changes in playback activity and media source states.

The API consists of two sections:

- The media browsing and playback interface
- The event interface

Media browsing and playback interface

The media browsing and playback interface defines the functions for connecting to `mm-player`, retrieving file information and metadata from media sources and media nodes, configuring tracksessions, and issuing playback commands. It also exposes the data types used by those functions for storing properties of media sources and media nodes.

The first function a client must call is *mm_player_connect()* (p. 57) to connect to `mm-player` and obtain a handle, which is required in all subsequent API calls. The client must then call *mm_player_open()* (p. 69) and pass in the name of a player to connect with. The player is used to carry out the browsing and playback actions.

To discover media content, the client must call *mm_player_get_media_sources()* (p. 64) to obtain a list of accessible media sources.

Before attempting any browsing or playback operation, the client should verify that the media source it's using supports that operation by examining the flag field that specifies its capabilities. This field is found in the `mmp_ms_t` (p. 47) structure that describes the media source.

The *mm_player_browse()* (p. 55) function retrieves information on the media nodes found on a media source. These media nodes may be folders or individual media files. To play media files, the client must first define a tracksession by calling *mm_player_create_trksession()* (p. 57) and then manage playback by issuing commands such as *mm_player_play()* (p. 70), *mm_player_stop()* (p. 77), and *mm_player_next()* (p. 68).

When it's finished browsing and playing media with `mm-player`, the client can disconnect by calling *mm_player_disconnect()* (p. 59).

Constants in `types.h`

Constants defined in `types.h` to specify the maximum size of data exchanged between the client library and the `mm-player` server.

Definitions in *types.h*

Preprocessor macro definitions for the `types.h` header file in the mmplayerclient library.

Definitions:

```
#define MAX_MSGSIZE KILO(64)
```

Maximum allowable size (64 kB) for messages exchanged between the client library and the server.

`#define MAX_PPSMSGSIZE KILO(32)`

Maximum allowable size (32 kB) for PPS data sent from the server to the client library.

`#define MMP_NAME_LEN 31`

Maximum length of a player's name.

Library:

`mmplayerclient`

Enumerations in `types.h`

Enumerations defined in `types.h` for listing the device types, file types, media source statuses, and playback modes supported by `mm-player`.

ms_browse_capability_e

Media source browsing operations.

Synopsis:

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

typedef enum {
      MS_BROWSE_CAPABILITY_METADATA = (0x00010000),
      MS_BROWSE_CAPABILITY_SEARCH =   (0x00020000),
      MS_BROWSE_CAPABILITY_BROWSE =   (0x00040000),
      MS_BROWSE_CAPABILITY_EXTENDED_METADATA = (0x00080000)
} ms_browse_capability_e;
```

Data:

`MS_BROWSE_CAPABILITY_METADATA`

You can retrieve metadata from media nodes.

`MS_BROWSE_CAPABILITY_SEARCH`

You can retrieve media nodes with metadata properties matching a search string.

`MS_BROWSE_CAPABILITY_BROWSE`

You can browse a media node within a media source.

`MS_BROWSE_CAPABILITY_EXTENDED_METADATA`

You can retrieve extended metadata (i.e., nonstandard properties) from media nodes.

Library:

`mmplayerclient`

Description:

Media source browsing operations. These constants indicate which bits to examine in the *capabilities* field of the `mmp_ms_t` (p. 47) structure when testing if a media source supports a given browsing operation.

ms_node_type_e

Media node types.

Synopsis:

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

typedef enum {
      MS_NTYPE_UNKNOWN = 0,
      MS_NTYPE_FOLDER,
      MS_NTYPE_AUDIO,
      MS_NTYPE_VIDEO,
      MS_NTYPE_RESERVED1,
      MS_NTYPE_PHOTO,
      MS_NTYPE_NUMBER
} ms_node_type_e;
```

Data:

`MS_NTYPE_UNKNOWN`

Unknown file category

`MS_NTYPE_FOLDER`

Folder

`MS_NTYPE_AUDIO`

Audio file

`MS_NTYPE_VIDEO`

Video file

`MS_NTYPE_RESERVED1`

Reserved for future use

`MS_NTYPE_PHOTO`

Photo file

`MS_NTYPE_NUMBER`

End-of-list identifier

Library:

`mmplayerclient`

Description:

The `ms_node_type_e` enumerated type defines the possible file types of media nodes.

ms_playback_capability_e

Media source playback operations.

Synopsis:

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

typedef enum {
      MS_PLAY_CAPABILITY_PLAY =         (0x00000001),
      MS_PLAY_CAPABILITY_PAUSE =        (0x00000002),
      MS_PLAY_CAPABILITY_NEXT =         (0x00000004),
      MS_PLAY_CAPABILITY_PREVIOUS =     (0x00000008),
      MS_PLAY_CAPABILITY_SEEK =         (0x00000010),
      MS_PLAY_CAPABILITY_SET_PLAYBACK_RATE = (0x00000020),
      MS_PLAY_CAPABILITY_SHUFFLE =      (0x00000040),
      MS_PLAY_CAPABILITY_REPEAT_ALL =   (0x00000080),
      MS_PLAY_CAPABILITY_REPEAT_ONE =   (0x00000100),
      MS_PLAY_CAPABILITY_REPEAT_NONE =  (0x00000200),
      MS_PLAY_CAPABILITY_STOP =         (0x00000400),
      MS_PLAY_CAPABILITY_JUMP =         (0x00000800),
      MS_PLAY_CAPABILITY_GET_POSITION = (0x00001000),
      MS_PLAY_CAPABILITY_PLAYER_MODE =  (0x10000000),
      MS_PLAY_CAPABILITY_DEVICE_MODE =  (0x20000000)
} ms_playback_capability_e;
```

Data:

`MS_PLAY_CAPABILITY_PLAY`

Playback is supported.

`MS_PLAY_CAPABILITY_PAUSE`

Playback can be paused.

`MS_PLAY_CAPABILITY_NEXT`

You can skip to the next track.

`MS_PLAY_CAPABILITY_PREVIOUS`

You can skip to the previous track.

`MS_PLAY_CAPABILITY_SEEK`

You can seek to a specific playback position.

`MS_PLAY_CAPABILITY_SET_PLAYBACK_RATE`

Playback speed can be adjusted.

`MS_PLAY_CAPABILITY_SHUFFLE`

Playback can be shuffled (i.e., randomized)

`MS_PLAY_CAPABILITY_REPEAT_ALL`

You can repeat all tracks in the same order.

`MS_PLAY_CAPABILITY_REPEAT_ONE`

You can repeat one track continuously.

`MS_PLAY_CAPABILITY_REPEAT_NONE`

You can disable repeating.

`MS_PLAY_CAPABILITY_STOP`

Playback can be stopped.

`MS_PLAY_CAPABILITY_JUMP`

You can jump to a different track.

`MS_PLAY_CAPABILITY_GET_POSITION`

You can retrieve the current playback position.

`MS_PLAY_CAPABILITY_PLAYER_MODE`

The mm-player server can manage tracksessions.

`MS_PLAY_CAPABILITY_DEVICE_MODE`

The plugin accessing the media source can manage tracksessions.

Library:

`mmplayerclient`

Description:

Media source playback operations. These constants indicate which bits to examine in the *capabilities* field of the `mmp_ms_t` (p. 47) structure when testing if a media source supports a given playback operation.

ms_status_e

Media source statuses.

Synopsis:

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

typedef enum {
      MS_STATUS_NOT_READY,
      MS_STATUS_READY,
      MS_STATUS_1STPASS,
      MS_STATUS_2NDPASS,
      MS_STATUS_3RDPASS
} ms_status_e;
```

Data:

MS_STATUS_NOT_READY

The media source isn't ready because it's been disconnected or its status can't be read

MS_STATUS_READY

The media source is ready, meaning it's connected and its status can be read

MS_STATUS_1STPASS

The file information from the media source has been synchronized (currently unused)

MS_STATUS_2NDPASS

The media metadata from the media source has been synchronized (currently unused)

MS_STATUS_3RDPASS

The playlist entry information for the media source has been synchronized (currently unused)

Library:

`mmplayerclient`

Description:

The `ms_status_e` enumerated type defines the possible media source statuses. The status tells clients whether a media source is connected and if so, how much of its media information has been synchronized with its database.

ms_type_e

Hardware types for media sources.

Synopsis:

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

typedef enum {
      MS_TYPE_HDD =       0x00000001,
      MS_TYPE_USB =       0x00000002,
      MS_TYPE_IPOD =      0x00000010,
      MS_TYPE_DLNA =      0x00000100,
      MS_TYPE_BLUETOOTH = 0x00001000,
      MS_TYPE_MTP =       0x00010000,
      MS_TYPE_UNKNOWN =   0x00100000
} ms_type_e;
```

Data:

`MS_TYPE_HDD`

Local drive

`MS_TYPE_USB`

USB storage device

`MS_TYPE_IPOD`

iPod

`MS_TYPE_DLNA`

DLNA device

`MS_TYPE_BLUETOOTH`

Bluetooth device

`MS_TYPE_MTP`

Device with MTP files (e.g., Android, Win7/8 phone)

`MS_TYPE_UNKNOWN`

Customized media source

Library:

`mmplayerclient`

Description:

The `ms_type_e` enumerated type defines the media source hardware types supported by `mm-player`.

repeat_e

Repeat modes.

Synopsis:

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

typedef enum {
      REPEAT_OFF = 0,
      REPEAT_ALL,
      REPEAT_ONE
} repeat_e;
```

Data:

`REPEAT_OFF`

No tracks will be repeated (playback will stop when the end of the active tracksession is reached)

`REPEAT_ALL`

All tracks will be repeated in the same order (playback will loop)

`REPEAT_ONE`

The current track will be continuously repeated

Library:

`mmplayerclient`

Description:

The `repeat_e` enumerated type defines the repeat modes supported by `mm-player`.

shuffle_e

Shuffle modes.

Synopsis:

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

typedef enum {
      SHUFFLE_OFF = 0,
      SHUFFLE_ON
} shuffle_e;
```

Data:

SHUFFLE_OFF

Shuffling is off; tracks will be played sequentially

SHUFFLE_ON

Shuffling is on; tracks will be played in a random order

Library:

`mmplayerclient`

Description:

The `shuffle_e` enumerated type defines the shuffle modes supported by `mm-player`.

status_e

Player status.

Synopsis:

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

typedef enum {
      STATUS_DESTROYED = 0,
      STATUS_IDLE,
      STATUS_PLAYING,
      STATUS_PAUSED,
      STATUS_STOPPED
} status_e;
```

Data:

STATUS_DESTROYED

Reserved for future use

STATUS_IDLE

The player is created but no tracksession is defined so playback is currently not possible

STATUS_PLAYING

A track is currently playing

STATUS_PAUSED

Playback is paused

STATUS_STOPPED

Playback is stopped, no track is selected, or an error has occurred

Library:

`mmplayerclient`

Description:

The `status_e` enumerated type defines the possible player statuses, which reflect the current playback support and activity. The initial status after a player has been created is `STATUS_IDLE`. Playback isn't possible until you call *mm_player_create_trksession()* to define a tracksession for the player, at which point its status changes to `STATUS_STOPPED`. The status returns to `STATUS_IDLE` when you destroy the tracksession.

A status of `STATUS_PAUSED` means that playback will resume at the same position if the client calls *mm_player_play()*. In contrast, the `STATUS_STOPPED` status means that playback will restart at the beginning of the current track if the client calls that last function.

The `STATUS_DESTROYED` status means that no state information could be read from the player. This status could be returned in an event's information if the connection to the player gets abruptly closed after the event is generated but before the API call to retrieve that event.

Data types in `types.h`

Data types defined in `types.h` for storing information describing media sources, media nodes, tracks, tracksessions, and playback state.

mmp_ms_t

A media source from the perspective of a client.

Synopsis:

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

typedef struct mmp_ms {
    int id;
    char *uid;
    char *name;
    char *view_name;
    ms_type_e type;
    ms_status_e status;
    uint64_t capabilities;
} mmp_ms_t;
```

Data:

int id

Unique ID of the media source

char *uid

Unique ID of the hardware device

char *name

Media source name

char *view_name

Name of the view configured for the device type of the media source

ms_type_e type

Hardware type

ms_status_e status

Media source status (i.e., ready or not ready)

uint64_t capabilities

A flag field indicating the supported browsing and playback operations

Library:

`mmplayerclient`

Description:

The `mmp_ms_t` structure stores media source information useful to a client. This information allows clients to distinguish one media source from the others and to know its current status and which operations it supports. The *mm_player_get_media_sources()* function returns an array of these structures, with each array element storing the information about a single media source.

Clients must pass in the media source ID (which can be read from the *id* field) to a identify a particular media source when calling the API functions for browsing or searching media, getting metadata, or creating tracksessions.

The *view_name* field indicates the view configured for the plugin used to access content on the media source. You can see this plugin setting in the *configuration file* (p. 30). The default configuration file uses two views: `synced` and `live`. These views determine which extended metadata fields can be retrieved with *mm_player_get_extended_metadata()* (p. 63). You can define additional views to suit your HMI's needs. For instance, you can define multiple views to represent the metadata on a device in different ways and then combine the device's hardware UID with specific `view_name` values to distinguish the views.

The flag field in *capabilities* indicates which browsing and playback operations are supported by the media source. To test if a particular operation is supported, you must examine the bit that corresponds to the operation's enumeration code in either `ms_browse_capability_e` (p. 39) or `ms_playback_capability_e` (p. 41). For example, to test if the media source supports jumping to another track during playback, you must examine the bit in *capabilities* that corresponds to the `MS_PLAY_CAPABILITY_JUMP` code.

mmp_ms_node_t

A media node.

Synopsis:

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

typedef struct mmp_ms_node {
    char *id;
    char *name;
    ms_node_type_e type;
    int count;
    int ms_id;
    ms_type_e ms_type;
} mmp_ms_node_t;
```

Data:

char *id

Unique ID of the media node

char *name

Name of the media node

ms_node_type_e type

Media node type

int count

Number of children contained in this node (`-1` means unknown)

int ms_id

ID of the media source on which the media node is located

ms_type_e ms_type

Device type of the media source

Library:

`mmplayerclient`

Description:

The `mmp_ms_node_t` structure stores information on media nodes. A media node is a container of media objects stored on a media source. This container can be an individual media file (i.e., a track) or a folder that stores other media nodes (as indicated by *type*). The value in *name* can be a filename, folder name, or metadata name, depending on the configuration of `mm-player`.

The *mm_player_get_trksession_tracks()*, *mm_player_browse()*, and *mm_player_search()* functions each return an array of `mmp_ms_node_t` structures, with each array element storing the information of a single media node. When calling the functions for browsing, getting metadata, or creating tracksessions, clients must pass in the media node ID (which can be read from the *id* field). To start browsing a media source whose folder structure is unknown, clients can specify the path of the root folder (/) in the call to *mm_player_browse()*. They can then read the IDs of the media nodes found in the root folder and perform other browsing operations to explore these nodes.

mmp_ms_node_metadata_t

Metadata associated with a media node.

Synopsis:

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

typedef struct mmp_ms_node_metadata {
    char *title;
    int duration;
    char *artwork;
    char *artist;
    char *album;
    char *genre;
    char *year;
    int width;
    int height;
    int disc;
    int track;
    char *reserved;
} mmp_ms_node_metadata_t;
```

Data:

char *title

Media file title

int duration

Track duration (in milliseconds)

char *artwork

URL of filepath for artwork (NULL if there's no artwork)

char *artist

Artist name

char *album

Album name

char *genre

Genre

char *year

Year of creation

int width

Width (in pixels)

int height

Height (in pixels)

int disc

Disc number (`-1` if not applicable)

int track

Track index (`-1` if not applicable)

char *reserved

Reserved for future use

Library:

`mmplayerclient`

Description:

The `mmp_ms_node_metadata_t` structure stores media node metadata, which consists of track creation and runtime details. Clients can display this information in the HMI to improve the user experience.

This structure is filled in by the *mm_player_get_metadata()* function. The exact metadata fields that get filled in depend on the media node's file type and the plugin used to retrieve the metadata (which varies with the media source's hardware type).

For instance, the *artwork* field is supported by the AVRCP plugin but not the POSIX plugin. Meanwhile, the *duration*, *artist*, and *album* fields can be populated for audio tracks but not for videos or photos. However, the *width* and *height* fields can be populated for videos and photos, but not for audio tracks.

mmp_state_t

Core mm-player state information.

Synopsis:

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

typedef struct mmp_state {
    shuffle_e shuffle_mode;
    repeat_e repeat_mode;
    status_e status;
    float rate;
} mmp_state_t;
```

Data:

shuffle_e shuffle_mode

Shuffle mode

repeat_e repeat_mode

Repeat mode

status_e status

Player status

float rate

Playback rate (i.e., the speed of playback)

Library:

`mmplayerclient`

Description:

The `mmp_state_t` structure stores the current settings for playback order and speed as well as the player's status, which reflects its current playback support and activity. This structure is filled in by the *mm_player_get_current_state()* function. If you change the repeat or shuffle mode or the playback speed, you can call this function and examine the values returned in the `mmp_state_t` structure to confirm that your change was applied.

mmp_track_info_t

Track information.

Synopsis:

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

typedef struct mmp_track_info {
    int index;
    uint64_t tsid;
    mmp_ms_node_t *media_node;
    mmp_ms_node_metadata_t *metadata;
} mmp_track_info_t;
```

Data:

int index

Position of the track within the tracksession

uint64_t tsid

ID of the associated tracksession

mmp_ms_node_t *media_node

Media node the track is contained in

mmp_ms_node_metadata_t *metadata

Track metadata

Library:

`mmplayerclient`

Description:

The `mmp_track_info_t` structure stores details on an individual track. This structure is filled in by the *mm_player_get_current_track_info()* function.

The “current” track is the track either actively being played or selected to be played next, based on the player's shuffle and repeat settings and the indexing of tracks at the media source. The *index* field indicates the playback position within the tracksession identified by *tsid*.

mmp_trksession_info_t

Basic tracksession information.

Synopsis:

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

typedef struct mmp_trksession {
    uint64_t tsid;
    int length;
} mmp_trksession_info_t;
```

Data:

uint64_t tsid

Tracksession ID

int length

Number of tracks within the tracksession

Library:

`mmplayerclient`

Description:

The `mmp_trksession_info_t` structure stores information on the player's active tracksession. This structure is filled in by the *mm_player_get_current_trksession_info()* function. In this release, each player can store only one tracksession at a time, so the tracksession object doesn't reference individual tracks. Instead, each `mmp_track_info_t` object (which stores information on a single track) stores the track's index (i.e., offset) in the active tracksession. You can retrieve the tracksession's tracks by passing in its ID (read from the *tsid* field) to the *mm_player_get_trksession_tracks()* function.

Functions in `mmplayerclient.h`

Functions defined in `mmplayerclient.h` for connecting to `mm-player` and to specific players, browsing media nodes, defining tracksessions, retrieving playback state information, and issuing playback commands.

mm_player_browse()

Browse a media node within a media source.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_browse( mmplayer_hdl_t *hdl,
                      const int media_source_id,
                      const char *media_node_id,
                      const int offset,
                      int *limit,
                      mmp_ms_node_t **media_nodes )
```

Arguments:

hdl

The `mm-player` connection handle

media_source_id

The ID of the media source to browse

media_node_id

The ID of the media node to browse

offset

The offset to start browsing from within the media node

limit

On input, the maximum number of items requested. On output, the number of items actually found (if the function succeeded).

media_nodes

The media nodes found within the media node being browsed

Library:

`mmplayerclient`

Description:

Browse the contents of the folder media node identified by *media_node_id* and located on the media source identified by *media_source_id*. To start browsing a new media source with an unknown directory structure, pass in "/" for *media_node_id* to indicate the root folder.

You can define *offset* to make `mm-player` browse media nodes starting from a certain index in the set of items contained in the media node being browsed. You can also define *limit* to restrict how many items can be returned.

The media nodes returned by this function can be folders (which can contain other media nodes) or individual media files such as audio tracks, videos, or photos (i.e., *leaf nodes*). The library allocates memory for the *media_nodes* array but it's the caller's responsibility to later deallocate that memory. Each array element stores information on a single media node found during browsing.

Returns:

`0` on success, `-1` on failure

mm_player_close()

Close the connection to a player.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_close( mmplayer_hdl_t *hdl,
                     const char *player_name )
```

Arguments:

hdl

The `mm-player` connection handle

player_name

The player to disconnect from

Library:

`mmplayerclient`

Description:

Close the connection to the player specified by *player_name*. If you want to use the same player again, you must call *mm_player_open()* (p. 69) again.

Returns:

`0` on success, `-1` on failure

mm_player_connect()

Connect to mm-player.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

mmplayer_hdl_t* mm_player_connect( int flags )
```

Arguments:

flags

Reserved for future use

Library:

`mmplayerclient`

Description:

Connect to `mm-player`. You must call this function before any other API functions. This function returns an `mm-player` handle, which is needed in all other function calls.

Returns:

A valid handle on success, NULL on failure (*errno* is set)

mm_player_create_trksession()

Create a tracksession from a media node.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_create_trksession( mmplayer_hdl_t *hdl,
                                 const int media_source_id,
                                 const char *media_node_id,
                                 int *limit,
                                 const int index,
                                 uint64_t *tsid )
```

Arguments:

hdl

The `mm-player` connection handle

media_source_id

The ID of the media source the base media node is stored on

media_node_id

The ID of the base media node

limit

On input, the maximum number of items to place in the tracksession. On output, the number of items actually placed in the tracksession (if the function succeeded).

index

The index of the tracksession item to be set as the current track

tsid

Currently, this is set to `0` because it's reserved for future use

Library:

`mmplayerclient`

Description:

Create a tracksession filled with tracks found within a "base" media node, which is identified by *media_node_id*. The media node must be located on the media source identified by *media_source_id*.

The items put into the new tracksession depend on the type of the base node. If it's a leaf node (i.e., an audio track, video, or photo), the tracksession contains only that media file. If it's a folder node, the tracksession contains its children (which can be media files or subfolders) up to the number of directory levels specified in the `recursion_depth` setting (for details, see "*Configuration file* (p. 30)"). For instance, suppose `recursion_depth` is `1`; in this case, the tracksession contains the media files from the base node folder but none of its subfolders or their media files. If this setting is `-1`, there's no limit to the directory depths of the items included in the tracksession.

You can define *limit* to restrict how many tracks get placed in the tracksession as well as *index* to set the current track (i.e., the track to be played first). For example, suppose you specify an index of `5`. The track at position `5` in the children of the base node then becomes the current track. Note that these two parameters apply only when the base node is a folder.

Returns:

`0` on success, `-1` on failure

mm_player_destroy_trksession()

Delete a tracksession.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_destroy_trksession( mmplayer_hdl_t *hdl,
                                  const uint64_t tsid )
```

Arguments:

hdl

The `mm-player` connection handle

tsid

The tracksession ID

Library:

`mmplayerclient`

Description:

Delete the tracksession specified by *tsid*. In this release, each player can have only one active tracksession, so after this call no tracksession will be available to the player and hence, playback won't be possible until a new tracksession is defined.

Returns:

`0` on success, `-1` on failure

mm_player_disconnect()

Disconnect from mm-player.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_disconnect( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Disconnect from `mm-player`. This function must be the last one you call. The handle in *hdl* is invalidated by this function and can't be used afterwards.

Returns:

`0` on success, `-1` if any resources could not be fully released

mm_player_get_current_state()

Get the player's state.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_current_state( mmplayer_hdl_t *hdl,
                                 mmp_state_t **state )
```

Arguments:

hdl

The `mm-player` connection handle

state

The player's current state

Library:

`mmplayerclient`

Description:

Get the state of the currently connected player. The library allocates memory for the structure referenced in *state* but it's the caller's responsibility to later deallocate that memory.

Returns:

`0` on success, `-1` on failure

mm_player_get_current_track_info()

Get information on the current track.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_current_track_info(
                        mmplayer_hdl_t *hdl,
                        mmp_track_info_t **track_info )
```

Arguments:

hdl

The `mm-player` connection handle

track_info

Information on the current track

Library:

`mmplayerclient`

Description:

Get information on the current track, which is the track either actively being played or selected to be played next. This information includes the track's metadata and its index (i.e., playback position) in the active tracksession. Client applications can then refresh their HMI display and deliver this up-to-date media information to users.

The library allocates memory for the structure referenced in *track_info* but it's the caller's responsibility to later deallocate that memory.

Returns:

`0` on success, `-1` on failure

mm_player_get_current_track_position()

Get the position of the current track.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_current_track_position(
                                  mmplayer_hdl_t *hdl,
                                  int *position )
```

Arguments:

hdl

The `mm-player` connection handle

position

The current position, in milliseconds from the start of the track

Library:

`mmplayerclient`

Description:

Get the position of the currently playing or paused track. The call fails if there's no active tracksession or track. We recommend retrieving the current track's position through the event interface instead of using this function.

Returns:

`0` on success, `-1` on failure

mm_player_get_current_trksession_info()

Get tracksession information.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_current_trksession_info(
                    mmplayer_hdl_t *hdl,
                    mmp_trksession_info_t **trksession_info )
```

Arguments:

hdl

The `mm-player` connection handle

trksession_info

Information on the current tracksession

Library:

`mmplayerclient`

Description:

Get information on the current tracksession. In this release, each player can store only one tracksession at a time, so the last tracksession created is always the “current” tracksession.

The library allocates memory for the structure referenced in *trksession_info* but it's the caller's responsibility to later deallocate that memory.

Returns:

`0` on success, `-1` on failure

mm_player_get_extended_metadata()

Get extended metadata associated with a media node.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_extended_metadata(
                             mmplayer_hdl_t *hdl,
                             const int media_source_id,
                             const char *media_node_id,
                             char *const keyv[],
                             char *valuev[] )
```

Arguments:

hdl

The `mm-player` connection handle

media_source_id

The media source ID

media_node_id

The media node ID

keyv

Array listing the metadata fields to retrieve

valuev

Array storing the values read from the requested metadata fields

Library:

`mmplayerclient`

Description:

Get extended metadata associated with the media node identified by *media_node_id* and located on the media source identified by *media_source_id*. Here, *extended metadata* refers to nonstandard metadata values that aren't returned by *mm_player_get_metadata()*, such as the URL of a media node.

The *mm_player_get_extended_metadata()* function retrieves values for the fields listed in *keyv* and populates *valuev* with references to the strings that store those values. The library allocates the memory for the strings, but the caller must provide sufficient memory in *valuev* for storing the references. The last member of *keyv* must be a NULL pointer. The size of *keyv* and *valuev* must be the same.

The default `mm-player` configuration defines two different views for reading extended metadata from a media source. You can see which view is configured for a given media source by:

- Reading the *view_name* field in the `mmp_ms_t` structure whose *id* field matches the value of *media_source_id*. The *mm_player_get_media_sources()* (p. 64) function returns an array of these structures, with each array element describing a particular media source.
- Examining the *configuration file* (p. 30), which lists the view name for each plugin used to access a specific device type. You must know the device type of your media source to know which plugin configuration determines the fields that you can read.

If the *view_name* value is "`synced`", you can retrieve the `folder_type` and `url` extended metadata fields (by listing them in *keyv*). If *view_name* is "`live`", you can retrieve only the `url` field.

The media node can be either a folder or a leaf node. However, `folder_type` applies only to folder nodes and `url` applies only to leaf nodes.

Returns:

`0` on success, `-1` on failure

mm_player_get_media_sources()

Get information on all connected media sources.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_media_sources( mmplayer_hdl_t *hdl,
                                 mmp_ms_t **media_sources,
                                 int *len )
```

Arguments:

hdl

The `mm-player` connection handle

media_sources

An array containing the connected media sources

len

The array size

Library:

`mmplayerclient`

Description:

Get information on all media sources connected to the system. The library allocates memory for the array that gets stored in *media_sources* but it's the caller's responsibility to later deallocate that memory. Each array element stores information on a single media source. This information includes the media source's name, hardware type, and supported browsing and playback operations. Client applications can display this information in the HMI to provide users with data describing the accessible media devices.

Returns:

`0` on success, `-1` on failure

mm_player_get_metadata()

Get metadata associated with a media node.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_metadata( mmplayer_hdl_t *hdl,
                        const int media_source_id,
                        const char *media_node_id,
                        mmp_ms_node_metadata_t **metadata )
```

Arguments:

hdl

The `mm-player` connection handle

media_source_id

The media source ID

media_node_id

The media node ID

metadata

The metadata associated with the media node

Library:

`mmplayerclient`

Description:

Get metadata associated with the media node specified by *media_node_id* and located on the media source identified by *media_source_id*. The media node must be a leaf node, that is, not a folder. The metadata retrieved includes track creation and playback details, which client applications can display in the HMI to provide users with useful media information.

The library allocates memory for the structure referenced in *metadata* but it's the caller's responsibility to later deallocate that memory.

Returns:

`0` on success, `-1` on failure

mm_player_get_trksession_tracks()

Get information on the media nodes associated with a tracksession.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_get_trksession_tracks(
                        mmplayer_hdl_t *hdl,
                        const uint64_t tsid,
                        const int offset,
                        int *limit,
                        mmp_ms_node_t **media_nodes )
```

Arguments:

hdl

The `mm-player` connection handle

tsid

The tracksession ID

offset

The tracksession offset to starting reading tracks from

limit

On input, the maximum number of items requested. On output, the number of items actually found (if the function succeeded).

media_nodes

The media nodes of the tracks contained in the tracksession

Library:

`mmplayerclient`

Description:

Get information on the media nodes associated with a tracksession. This function allocates the necessary memory and populates the *media_nodes* array, with each array element storing information on a single track in the tracksession identified by *tsid*. It's the caller's responsibility to later deallocate the array memory.

You can restrict which tracks get retrieved by setting the *offset* and *limit* parameters. The *offset* parameter makes `mm-player` retrieve media nodes for only those tracks at or beyond a certain offset (i.e., playback position) within the tracksession. The *limit* parameter restricts how many tracks can be retrieved.

Returns:

`0` on success, `-1` on failure

mm_player_jump()

Jump to a new track in the tracksession.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_jump( mmplayer_hdl_t *hdl,
                    const int index )
```

Arguments:

hdl

The `mm-player` connection handle

index

The tracksession position of the next track to play

Library:

`mmplayerclient`

Description:

Jump to a new track in the tracksession. This function sets the current track (i.e., the tracksession item selected for playback) to the track specified by *index*.

If playback is active when this function is called, the player starts playing the track specified by *index*. If playback isn't active, the track specified by *index* will be played when playback resumes.

Returns:

`0` on success, `-1` on failure

mm_player_next()

Skip to the next track.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_next( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Skip to the next track. The track considered as the "next track" depends on the shuffle and repeat mode settings.

When the repeat mode is `REPEAT_ONE`, the next track is always the current track (because it's set to repeat continuously).

When the repeat mode is REPEAT_ALL, the next track is the track immediately following the current track in either the sequential playback list (if shuffling is off) or in the randomized list (if shuffling is on). If the current track is the last track in the list, the next track is the first track in the list (because playback is looped).

When the repeat mode is REPEAT_OFF, the next track is selected in a similar manner based on the shuffle mode, except if the current track is the last track in the list, in which case playback stops because there is no next track.

Returns:

0 on success, -1 on failure

mm_player_open()

Open a connection to a player.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_open( mmplayer_hdl_t *hdl,
                    const char *player_name,
                    int oflag )
```

Arguments:

hdl

The mm-player connection handle

player_name

The player to connect to

oflag

Flags specifying the status and access modes. This field is unused and should be set to 0.

Library:

mmplayerclient

Description:

Open a connection to the player specified by *player_name*. If the player doesn't exist, it gets created. The player is the mechanism that carries out the browsing and playback actions, so all subsequent API commands using the same mm-player handle will be directed to this player.

Returns:

`0` on success, `-1` on failure

mm_player_pause()

Pause playback.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_pause( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Pause playback. This function changes the player status to STATUS_PAUSED but maintains the current playback position. This way, you can resume playback at the exact position at which you paused it.

Returns:

`0` on success, `-1` on failure

mm_player_play()

Start playback.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_play( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Start playback. This function changes the player status to `STATUS_PLAYING`. This status setting remains in effect until you pause or stop playback, or the end of the tracksession is reached and repeating is disabled.

The track that begins playing is the one selected as the "current" track in the active tracksession. When this track finishes playing, the player chooses a new track to play based on the shuffle and repeat mode settings. For more details, see *mm_player_repeat()* (p. 72) and *mm_player_shuffle()* (p. 76).

At any time during playback, you can seek to a new position in the current track by calling *mm_player_seek()* (p. 74) or change the current track by calling *mm_player_jump()* (p. 67).

Returns:

`0` on success, `-1` on failure

mm_player_previous()

Skip to the previous track.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_previous( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Skip to the previous track. The track considered as the "previous track" depends on the shuffle and repeat mode settings.

When the repeat mode is `REPEAT_ONE`, the previous track is always the current track (because it's set to repeat continuously).

When the repeat mode is `REPEAT_ALL`, the previous track is the track immediately preceeding the current track in either the sequential playback list (if shuffling is off) or in the randomized list (if shuffling is on). If the current track is the first track in the list, the previous track is the last track in the list (because playback is looped).

When the repeat mode is `REPEAT_OFF`, the previous track is selected in a similar manner based on the shuffle mode, except if the current track is the first track in the list, in which case playback stops because there is no previous track.

Returns:

`0` on success, `-1` on failure

mm_player_repeat()

Set the repeat mode.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_repeat( mmplayer_hdl_t *hdl,
                      const repeat_e mode )
```

Arguments:

hdl

The `mm-player` connection handle

mode

The new repeat mode setting

Library:

`mmplayerclient`

Description:

Set the repeat mode. This function allows you to repeatedly play an individual track or a sequence of tracks.

The `REPEAT_ONE` repeat mode causes the player to play the same track continuously until you either stop playback or skip to another track. A repeat mode of `REPEAT_ALL` makes the player play all the tracks in the active tracksession and then loop back to the beginning of the tracksession. The playback order is either sequential (when shuffling is off) or random (when shuffling is on). If the repeat mode is `REPEAT_OFF`, the player plays all the tracks exactly once but stops when it reaches the end of the tracksession.

By default, repeating is disabled, meaning the repeat mode is `REPEAT_OFF` when a player is created.

Returns:

`0` on success, `-1` on failure

mm_player_search()

Search for media nodes with metadata properties matching a search string.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_search( mmplayer_hdl_t *hdl,
                      const int media_source_id,
                      const char *filter,
                      const char *search_term,
                      const int offset,
                      int *limit,
                      mmp_ms_node_t **media_nodes )
```

Arguments:

hdl

The `mm-player` connection handle

media_source_id

The ID of the media source to search

filter

A JSON-formatted string listing the metadata fields to examine for values matching the search string (can be NULL to examine all fields)

search_term

The search string

offset

The offset to start searching from within the root folder of the media source

limit

On input, the maximum number of items requested. On output, the number of items actually found (if the function succeeded).

media_nodes

The media nodes found in the media source that have metadata matching the search string

Library:

`mmplayerclient`

Description:

Search the media source identified by *media_source_id* for media nodes with metadata properties matching a search string. The *filter* parameter lets you list which properties are examined. You can set this field to NULL to examine all properties. Otherwise, this parameter must reference a JSON-formatted string that lists the properties between square brackets:

```
["artist","album","genre","song","video"]
```

This sample string shows all available filter fields. When assigning such a string to a `const char*` variable in C code, be sure to escape each double quote within the string by using a backslash (\).

The *search_term* parameter contains the text to match; regular expressions or wildcards aren't supported so it must be an exact match.

You can set *offset* to make `mm-player` search media nodes starting from a certain index within the set of child items (i.e., contained media nodes) in the root of the media source being searched. You can also set *limit* to restrict how many child items can be returned.

The library allocates memory for *media_nodes* but it's the caller's responsibility to later deallocate that memory.

Returns:

`0` on success, `-1` on failure

mm_player_seek()

Seek to a position in the current track.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_seek( mmplayer_hdl_t *hdl,
                    const int position )
```

Arguments:

hdl

The `mm-player` connection handle

position

The new track position (in milliseconds)

Library:

`mmplayerclient`

Description:

Seek to a position in the current track. The value in *position* is the number of milliseconds from the start of the track (e.g., 2500).

Returns:

`0` on success, `-1` on failure

mm_player_set_playback_rate()

Set the playback rate.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_set_playback_rate( mmplayer_hdl_t *hdl,
                                 const float rate )
```

Arguments:

hdl

The `mm-player` connection handle

rate

The new playback rate, relative to a normal rate of `1.0`

Library:

`mmplayerclient`

Description:

Set the playback rate (speed). The floating-point value in *rate* is relative to a normal rate of `1.0`. A value of `0` pauses playback.

Returns:

`0` on success, `-1` on failure

mm_player_shuffle()

Set the shuffle mode.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_shuffle( mmplayer_hdl_t *hdl,
                       const shuffle_e mode )
```

Arguments:

hdl

The `mm-player` connection handle

mode

The new shuffle mode setting

Library:

`mmplayerclient`

Description:

Set the shuffle mode. This function lets you enable and disable randomized playback.

The shuffle mode setting determines which of two lists the player uses to select a new track for playback when the current track finishes playing. When the mode is `SHUFFLE_ON`, the player uses a randomized track list, which indexes tracks in an order different from their order in the media source. For example, when the track listed as number 2 on its album finishes playing, the next track played could be any other track on the album (including the track listed as number 3). When the mode is `SHUFFLE_OFF`, the player uses the sequential track list, which reflects the track order in the media source. In this case, when track number 2 finishes playing, track number 3 will play next.

When you call this function with *mode* set to `SHUFFLE_ON`, the player generates a new randomized playback list. So you can keep randomized playback enabled and just change to a different random order by calling this function multiple times with this mode setting.

By default, shuffling is disabled, meaning the shuffle mode is `SHUFFLE_OFF` when a player is created.

Returns:

`0` on success, `-1` on failure

mm_player_stop()

Stop playback.

Synopsis:

```
#include <mmplayer/mmplayerclient.h>

int mm_player_stop( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

The `mm-player` connection handle

Library:

`mmplayerclient`

Description:

Stop playback. This function changes the player status to `STATUS_STOPPED` and resets the playback position to the beginning of the current track.

Returns:

`0` on success, `-1` on failure

Event interface

The event interface enumerates the `mm-player` event types, defines the data structures that store event information, and defines the functions for processing `mm-player` events.

Events provide a practical mechanism for detecting changes in media sources and playback status. By using the event interface functions, you can receive notifications of such changes from `mm-player` instead of constantly polling it for state information and manually examining the retrieved data to detect those changes.

Enumerations in `events.h` and `types.h`

Enumerations defined in `events.h` and `types.h` for `mm-player` event types, media source and tracksession event types, and error severity levels.

ms_error_e

Media source error severity levels.

Synopsis:

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

typedef enum {
      MS_ERROR_RECOVERABLE = 0,
      MS_ERROR_NONRECOVERABLE
} ms_error_e;
```

Data:

MS_ERROR_RECOVERABLE

The error is recoverable

MS_ERROR_NONRECOVERABLE

The error is not recoverable

Library:

`mmplayerclient`

Description:

Media source error severity levels. For `MMP_EVENT_ERROR` events, the event information returned to the client includes an `mmp_event_error` structure with its *type* field set to one the `MS_ERROR_*` values.

ms_event_e

Media source event types.

Synopsis:

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

typedef enum {
      MS_EVENT_ADDED = 0,
      MS_EVENT_REMOVED,
      MS_EVENT_UPDATED
} ms_event_e;
```

Data:

MS_EVENT_ADDED

A media source has been connected

MS_EVENT_REMOVED

A media source has been disconnected

MS_EVENT_UPDATED

The status of a connected media source has changed

Library:

`mmplayerclient`

Description:

Media source event types. For `MMP_EVENT_MEDIASOURCE` events, the event information returned to the client includes an `mmp_event_mediasource` structure with its *type* field set to one the `MS_EVENT_*` values.

mmp_event_type_e

The mm-player event types.

Synopsis:

```
#include <mmplayer/event.h>

typedef enum mmp_event_type {
      MMP_EVENT_NONE = 0,
      MMP_EVENT_ERROR,
      MMP_EVENT_MEDIASOURCE,
      MMP_EVENT_STATE,
      MMP_EVENT_TRACKSESSION,
      MMP_EVENT_TRACK,
      MMP_EVENT_TRACKPOSITION,
      MMP_EVENT_OTHER
} mmp_event_type_e;
```

Data:

MMP_EVENT_NONE

No pending events

MMP_EVENT_ERROR

Browsing or playback has stopped due to an error

MMP_EVENT_MEDIASOURCE

State change for the media source

MMP_EVENT_STATE

State update (shuffle mode, repeat mode, status, or playback speed has changed)

MMP_EVENT_TRACKSESSION

Tracksession update (index, tsid, or the track items have changed)

MMP_EVENT_TRACK

Metadata update (msid, media node, metadata)

MMP_EVENT_TRACKPOSITION

Position update

MMP_EVENT_OTHER

None of the above, but something has changed; you can typically ignore this event type

Library:

`mmplayerclient`

Description:

The `mm-player` event types. For all events, the *type* field in the `mmp_event_t` structure is set to one of the `MMP_EVENT_*` values.

trksession_event_e

Tracksession event types.

Synopsis:

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

typedef enum {
      TRKSESSION_EVENT_CREATED = 0,
      TRKSESSION_EVENT_DESTROYED,
      TRKSESSION_EVENT_APPENDED
} trksession_event_e;
```

Data:

TRKSESSION_EVENT_CREATED

A tracksession has been created

TRKSESSION_EVENT_DESTROYED

A tracksession has been destroyed

TRKSESSION_EVENT_APPENDED

A tracksession has been appended with additional tracks

Library:

mmplayerclient

Description:

Tracksession event types. For MMP_EVENT_TRACKSESSION events, the event information returned to the client includes an mmp_event_trksession structure with its *type* field set to one the TRKSESSION_EVENT_* values.

Data types in events.h

Data types defined in event.h for storing event information delivered to clients.

mmp_event

The mm-player event information.

Synopsis:

```
#include <mmplayer/event.h>

typedef struct mmp_event {
    mmp_event_type_e type;
    repeat_e repeat;
    shuffle_e shuffle;
    status_e status;
    float rate;
    union mmp_event_details details;
    const strm_dict_t *data;
    const char *objname;
    void *usrdata;
} mmp_event_t;
```

Data:

mmp_event_type_e type

Event type

repeat_e repeat

The player's repeat mode

shuffle_e shuffle

The player's shuffle mode

status_e status

The player's status

float rate

Playback rate

union mmp_event::mmp_event_details details

Additional details that vary by event type

const strm_dict_t *data

The set of `mm-player` properties reported by the event, stored in a dictionary object. This field is NULL when no set of properties exists; for example, if a media source has just been connected and no information has been read yet.

const char *objname

The name of the internal `mm-player` object associated with this event

void *usrdata

The user data associated with the dictionary; this field is always NULL because it's currently unused

Library:

`mmplayerclient`

Description:

The `mmp_event_t` structure is returned by *mmp_event_get()* (p. 86) and stores information on the latest event. For all events, the structure contains the event type and the latest player state information (e.g., repeat mode, playback status). For all event types except `MMP_EVENT_NONE` and `MMP_EVENT_OTHER`, the structure contains additional, type-specific information in the *details* and *data* fields. For instance, for `MMP_EVENT_MEDIASOURCE` events, each of these fields stores the type of the media source event (e.g., a media source connection or a disconnection) and information on the media source affected by the event.

mmp_event::details

Type-specific mm-player event details.

Synopsis:

```
#include <mmplayer/event.h>

union mmp_event_details {

 struct mmp_event_state {
  status_e   oldstatus;
  float      oldrate;
  repeat_e   oldrepeat;
  shuffle_e  oldshuffle;
 } state;

 struct mmp_event_trksession {
  trksession_event_e type;
  int        length;
  uint64_t   tsid;
 } trksession;

 struct mmp_event_error {
  ms_error_e type;
 } error;

 struct mmp_event_track {
  int        index;
  uint64_t   tsid;
  mmp_ms_node_t *media_node;
  mmp_ms_node_metadata_t *metadata;
 } track;

 struct mmp_event_trkpos {
  int        position;
 } trkpos;

 struct mmp_event_mediasource {
  ms_event_e type;
  mmp_ms_t   *mediasource;
 } mediasource_info;
} details;
```

Data:

state

Used when `mmp_event.type` is `MMP_EVENT_STATE`.

The `mmp_event_state` structure has these members:

`status_e` ***oldstatus***

The player's status before the event

`float` ***oldrate***

The playback rate (speed) before the event

`repeat_e` ***oldrepeat***

The player's repeat mode before the event

`shuffle_e` ***oldshuffle***

The player's shuffle mode before the event

trksession

Used when `mmp_event.type` is `MMP_EVENT_TRACKSESSION`.

The `mmp_event_trksession` structure has these members:

`trksession_event_e` ***type***

The tracksession event type

`int` ***length***

The number of tracks within the tracksession

`uint64_t` ***tsid***

The tracksession ID

error

Used when `mmp_event.type` is `MMP_EVENT_ERROR`.

The `mmp_event_error` structure has these members:

`ms_error_e` ***type***

The severity level of the media source error

track

Used when `mmp_event.type` is `MMP_EVENT_TRACK`.

The `mmp_event_track` structure has these members:

`int` ***index***

The track's position within the tracksession

`uint64_t` ***tsid***

The ID of the associated tracksession

`mmp_ms_node_t` ****media_node***

A reference to the structure that describes the media node associated with the track

`mmp_ms_node_metadata_t` ****metadata***

A reference to the structure that stores the track's metadata

trkpos

Used when `mmp_event.type` is `MMP_EVENT_TRACKPOSITION`.

The `mmp_event_trkpos` structure has these members:

`int` ***position***

The track's new position in the tracksession

mediasource_info

Used when `mmp_event.type` is `MMP_EVENT_MEDIASOURCE`.

The `mmp_event_mediasource` structure has these members:

`ms_event_e` ***type***

The media source event type

`mmp_ms_t` ****mediasource***

A reference to the structure that describes the media source

Library:

`mmplayerclient`

Description:

Type-specific `mm-player` event details. Only one structure within the *details* field can be defined at a time. The structure that's defined depends on the event type.

Applications must read the *mmp_event.type* field to learn the event type so they can properly parse the contents of the *details* field.

mmp_event::data

The properties reported by the event, stored in dictionary format.

Description:

All the `mm-player` properties reported by the event, represented as a dictionary object. This field provides an alternative mechanism for reading event properties.

For all event types, the names of the dictionary keys match the names of the equivalent fields in the structure defined in *mmr_event::details*. The exception is any field with a name consisting of multiple words, such as *mediasource* (found in the `mmp_event_mediasource` structure when the event type is `MMP_EVENT_MEDIASOURCE`). The equivalent dictionary key contains an underscore (_) between the individual words (i.e., `media_source` is the correct key).

To look up values in a `strm_dict_t` dictionary object by key name, see the *strm_dict_find_value()* function in the *Multimedia Renderer Developer's Guide*.

Functions in `events.h`

Functions defined in `events.h` for processing `mm-player` events.

mmp_event_get()

Get the next available event.

Synopsis:

```
#include <mmplayer/event.h>

const mmp_event_t* mmp_event_get( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

A player handle

Library:

`mmplayerclient`

Description:

Get the next available event. This function returns an `mmp_event_t` (p. 81) structure populated with the event details. Typically, you would call this function within an event-processing loop, after calling *mmp_event_wait()* (p. 87).

The data in the returned structure is valid only until the next *mmp_event_get()* call. If you want to keep the data longer, copy the `mmp_event_t` contents into other program variables, cloning any `strm_string_t` fields within the structure.

If an event occurs, causing *mmp_event_wait()* to return, but the event's data gets deleted by `mm-player` before you call *mmp_event_get()*, the latter function will return the `MMP_EVENT_NONE` event. This event does *not* signify an error but instead that the previous event is no longer available. Applications must gracefully handle the `MMP_EVENT_NONE` event, preferrably by ignoring it.

Returns:

A pointer to an event, or NULL on error

mmp_event_wait()

Wait until an event is available.

Synopsis:

```
#include <mmplayer/event.h>

int mmp_event_wait( mmplayer_hdl_t *hdl )
```

Arguments:

hdl

A player handle

Library:

`mmplayerclient`

Description:

Wait for an `mm-player` event. This function usually blocks until an event occurs, at which point it unblocks and you can call *mmp_event_get()* (p. 86) to get the event details.

Typically, you call *mmp_event_wait()* within an event-processing loop, right before you call *mmr_event_get()*.

Returns:

`0` on success, or `-1` on error

Index