Multimedia Renderer 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: March 25, 2015

Contents

About This Guide....7
Typographical conventions....8
Technical support....10

Chapter 1: Multimedia Renderer: Capabilities and Architecture....11
Supported media categories....12
Abstraction layers....13
Contexts....14
Outputs....15
Inputs....16
Plugins....17

Chapter 2: Using the Multimedia Renderer....19
Starting the multimedia renderer....20
Configuration file for `mm-renderer`....20
Command line for `mm-renderer`....22
Working with contexts....24
Closing context handles....24
Defining Parameters....26
Playing media....27
Play states....27
Play speed....27
Seeking to positions....28
Managing video windows....28
Recording audio data....33
PPS objects....35
Context state....35
Play state, warnings, and errors....36
Input metadata....37
Playlist window....37
Supported file and MIME types....38

Chapter 3: Multimedia Renderer API....39
Connection management....40
mmr_connect()....41
mmr_connection_t....42
mmr_disconnect()....43
Context management....44
mmr_command_send()....45
mmr_context_close()....46
mmr_context_create()....47
mmr_context_destroy()....49

mmr_context_open()....50
mmr_context_parameters()....52
mmr_context_t 54
Error information....55
mm_error_code_t....56
mmr_error_info()....60
mmr_error_info_t....61
Events....63
mmr_event_arm()....64
mmr_event_data_set()....66
mmr_event_get()....68
mmr_event_t....70
mmr_event_type_t....78
mmr_event_wait()....80
mmr_state_t....82
Input configuration....83
mmr_input_attach()....84
mmr_input_detach()....88
mmr_input_parameters()....89
mmr_track_parameters()....92
Output configuration....95
mmr_output_attach()....96
mmr_output_detach()....100
mmr_output_parameters()....101
Playback control....105
mmr_list_change()....106
mmr_play()....108
mmr_seek()....109
mmr_speed_set()....110
mmr_stop()....112

Chapter 4: Dictionary Object API....113
strm_dict_clone()....114
strm_dict_compare()....115
strm_dict_destroy()....116
strm_dict_find_index()....117
strm_dict_find_rstr()....118
strm_dict_find_value()....119
strm_dict_index_delete()....120
strm_dict_key_delete()....121
strm_dict_key_get()....122
strm_dict_key_rstr()....123
strm_dict_new()....124
strm_dict_set()....125
strm_dict_set_rstr()....127
strm_dict_size()....129
strm_dict_subtract()....130

strm_dict_t....131
strm_dict_value_get()....133
strm_dict_value_rstr()....134
strm_string_alloc()....135
strm_string_clone()....136
strm_string_destroy()....137
strm_string_get()....138
strm_string_make()....139
strm_string_modify()....140
strm_string_t....141

Index....143

About This Guide

The *Multimedia Renderer Developer's Guide* is intended for developers who want to write multimedia applications that use the **mm-renderer** service to control playback.

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

To find out about:	Go to:
The main features of the **mm-renderer** service	*Multimedia Renderer: Capabilities and Architecture* (p. 11)
The setup tasks required to launch **mm-renderer**	*Starting the multimedia renderer* (p. 20)
Using contexts to define media flows	*Working with contexts* (p. 24)
The steps required to play media	*Playing media* (p. 27)
Connecting to **mm-renderer** and managing contexts to control playback	*Multimedia Renderer API* (p. 39)
Working with dictionary objects to define media properties	*Dictionary Object API* (p. 113)

Typographical conventions

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

The following table summarizes our conventions:

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

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

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

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Multimedia Renderer: Capabilities and Architecture

The multimedia rendering component, **mm-renderer**, allows multimedia applications to play audio and video media from files and devices.

The **mm-renderer** service provides mechanisms for:

- specifying the set of media to play
- issuing playback control commands
- retrieving the current playback status
- requesting notifications when the status changes
- providing dynamic metadata (such as position in a track or playlist) for some media content

The multimedia renderer API allows you to control media playback and recording and to monitor media operations by receiving events. To examine the system data used by **mm-renderer**, use the Persistent Publish/Subscribe (PPS) service. For more information, see the "*PPS objects* (p. 35)" section or the *PPS Developer's Guide*.

The **mm-renderer** service can play media content independently because it directly reads the specified input files without relying on information in databases. You may run **mm-sync** to synchronize media metadata with databases so that your applications can display up-to-date information, but this activity isn't necessary for playing media with **mm-renderer**.

Supported media categories

The **mm-renderer** service supports playback of tracks and playlists. The content can be read from local files, HTTP streams, or database queries (for playlists only). The media category is indicated in the input URL given to **mm-renderer**.

For the current list of supported device types, filesystems, codecs, and video formats, see the *Release Notes* for the QNX SDK for Apps and Media.

Sources for audio and video tracks

A *track* is an audio or video file such as an MP3 or MP4 file. You can play tracks from these sources:

Files

To play media content from files, specify the path (in POSIX format) of an audio or video file in the input URL, with or without a `file:` or `http:` prefix. You can name a dynamically growing file using the `file2b:` prefix, which lets you play content that's downloading.

HTTP streams

To play media content from HTTP sources, specify an HLS source or another type of HTTP stream in an input URL starting with `http:` or `https:`. The **mm-renderer** service supports cookies, SSL, and authentication, which enables secure playback of HTTP streams.

Audio capture devices

Use an input URL that either starts with `snd:` and lists the device path or that starts with `audio:` and names a supported audio device. For both URL types, you can configure several parameters such as the sampling rate, number of channels, and number of bits per sample.

More details on the options available with any of these URL types and the sources they refer to are given in the *mmr_input_attach()* (p. 84) description. The same reference also explains how you can treat a single track as its own playlist, using the `autolist` input type.

Supported playlist types

A *playlist* is a list of track URLs. The following types of playlists are supported:

M3U files

The input URL must be a full path (with or without a `file:` or `http:` prefix) of a file with an M3U extension. In its simplest form, an M3U file is a plain-text file containing the pathnames or URLs of the tracks to play (one per line). Playlists for HTTP Live Streaming (HLS) are supported; these may have M3U and M3U8 file extensions.

SQL queries

The input URL must be of the form `sql:`*database*`?query=`*querystring*, where *database* is a full path to a database file, and *querystring* is an SQL query that returns a column of track names.

The *mmr_input_attach()* description gives full details on *the format of playlist URLs* (p. 86).

Abstraction layers

The multimedia renderer uses a layered architecture to process playback commands and manage the flow of media content from input to output.

The **mm-renderer** architecture consists of three abstraction layers:

- The *messaging layer* decodes client messages and delivers them to *contexts*. A context is an object capable of playing one piece of input media at a time on a set of output devices. The input can contain both audio and video signals (e.g., a movie), and the set of outputs can consist of both audio and video devices (e.g., speakers and a screen).
- The *context management layer*:
  - keeps track of the outputs attached to each context
  - maps each output to the engine plugins that can support that output type
  - selects the appropriate engine plugins and attaches them to the context
  - preserves the context state between detaching and reattaching inputs
  - publishes the context state through the PPS service
  - delivers client requests (e.g., *play*) to engine plugins
- The *plugin management layer* keeps track of all available plugins.

Contexts

The **mm-renderer** service provides *contexts*, each of which can play a stream of media content concurrently with and independently of other contexts. Each context can direct output to a different set of hardware devices or files, creating independent zones of operation.

The operations that are available for a context at a particular time depend on the input and outputs attached to it. For example, changing playlists won't work unless your input type is a playlist, and seeking to a new track position doesn't apply for some input streams (e.g., radio stations) or devices (e.g., microphones). You can use a context for operations other than playing, by setting its output appropriately. For instance, you can record rather than play an audio stream by setting a context's output to a file and its input to an audio capture device (i.e., a microphone).

Your application must connect to the **mm-renderer** service before it can create a context. When your application creates a context, the context has a unique name but no other properties are set. For subsequent operations, your application accesses the context through the handle returned by **mm-renderer** when it created the context. Depending on its configuration, **mm-renderer** may behave differently when you close context handles.

You can control properties of the context's operation (e.g., audio volume) by attaching *parameters* to the context, its input, or each of its outputs (for details, see "*Parameters* (p. 26)").

Outputs

Each context has to have one or more outputs attached before it can play anything, so that it can determine where to send the content.

The output can be:

- a file
- an audio device
- a video device

Outputs need to be attached before the input. This is because some engine plugins may determine whether to play an input based on what kind of outputs are attached, and may not support detaching and reattaching outputs after the input is connected.

Inputs

Each context has to have one input attached, so it knows what to play.

When an input is attached, the context management layer selects the appropriate engine plugin and attaches it to the context. Your application has to identify the input type for **mm-renderer** because the service doesn't automatically detect the type of the attached input.

The input type determines how **mm-renderer** responds to certain playback requests, such as seeking to a track position or changing playlists. Which input types are supported depends on the configuration of **mm-renderer**; however, the playback behavior for a given input type does *not* depend on the configuration.

Although an input may be attached to more than one context, **mm-renderer** doesn't detect or manage conflicting playback operations—your application must manage potential playback conflicts.

Plugins

Engine plugins are the components used by **mm-renderer** to process media data read from the input before directing that data to the attached outputs.

The implementation of the engine plugins is invisible to **mm-renderer** and its clients. The context management layer selects the appropriate plugin based on the types of the input and outputs attached to the context and on the rating that each plugin gives itself for the specified input and outputs.

Chapter 2
Using the Multimedia Renderer

The multimedia renderer is a connection-based service that controls the playback of media read from an input and directed to one or many outputs.

Before calling any API functions to start and control playback, you must start **mm-renderer** and your application must connect to it by calling *mmr_connect()* (p. 41). After you're connected, you can:

1. Create contexts for managing individual media content flows.
2. Attach inputs and outputs to contexts.
3. Configure parameters to optimize **mm-renderer** to play certain audio content and to set authentication and proxy settings if reading content from HTTP streams.
4. Issue media playback commands.

When you're finish using **mm-renderer**, you should disconnect from the service by calling *mmr_disconnect()* (p. 43).

Starting the multimedia renderer

The multimedia renderer service is controlled with the **mm-renderer** command utility. Before starting the service, you must prepare a configuration file and set up your PPS state directory.

To start **mm-renderer**:

1. Create and save, in the **/etc/mm/** directory, the configuration file (named **mm-renderer.conf**) to use.

   For a sample configuration file, see "*Configuration file for mm-renderer* (p. 20)".

2. In a QNX Neutrino terminal, enter `pps` to start PPS as a background process.

   PPS creates a root directory (**/pps** by default) to store the PPS configuration objects, which are text files that provide state, error, and playlist information on the multimedia renderer's setup.

3. Enter `mkdir -p /pps/services/multimedia/renderer` to create the root directory for holding the PPS objects used by **mm-renderer**.

   For an explanation of the PPS objects used by **mm-renderer**, see "*PPS objects* (p. 35)".

4. Enter `mm-renderer` followed by any desired options to start the multimedia renderer service.

   For debugging purposes, you should start **mm-renderer** with `-vvvvvvv` options to get verbose output. The `-v` option is cumulative, with each `v` adding a level of verbosity, up to seven levels.

   For an explanation of all command-line options, see "*Command line for mm-renderer* (p. 22)".

The PPS and multimedia renderer services are running. Client applications can now use the **mm-renderer** API to control playback.

Configuration file for `mm-renderer`

The multimedia renderer configuration file specifies the plugins to load and the default values for plugin parameters. The path of this file is **/etc/mm/mm-renderer.conf**, and the file must be defined before you start **mm-renderer**.

A single configuration file is used for all instances of **mm-renderer**. Thus, all running instances of the service have the same plugins available, with the same default parameters.

After starting, **mm-renderer** does not support "on-the-fly" configuration changes. To modify the configuration, you must shut down **mm-renderer**, update its configuration file, and restart the service.

The configuration file is a text file that gets parsed by **mm-renderer** to load and configure the plugins used in playback. Blank lines are ignored, as is any leading or trailing white space. Lines specifying parameters are in the form *key=value*. Unknown parameter types are ignored and so they can be made into comments. For example, you can enter `My Comment Goes Here` on a line and **mm-renderer** will consider it to be an unsupported parameter and so will ignore that line when parsing the file. However, we recommend using the number sign ('#') to clearly indicate the start of any comments.

The file is divided into sections, each of which configures one plugin. Sections begin with the `[plugin]` keyword on its own line. The name of the plugin library file is specified on the next line, with the `dll` key. Default values to any parameters supported by the plugin may be specified on the lines that follow.

Suppose you want to use the audio/video player routing plugin, and override the default status update interval of 1000 ms with an update interval of half that time. You would then write a section in the configuration file as follows:

```
# Configure the audio/video player routing plugin
[plugin]
dll=mmr-mmf-routing.so
# Set a half-second (500ms) interval between updates
updateinterval=500
```

Any of the default values set in the configuration file can be overridden by client applications by setting the parameter through the API.

The following plugins and associated parameters are supported:

MMF audio recorder routing plugin

Library file: **mmr-mmfrip-routing.so**

There are no parameters for this plugin.

Playlist engine plugin

Library file: **mmr-playlist-engine.so**

This plugin supports the "`playlist`" and "`autolist`" input types, and has these parameters:

queue_max

The maximum number of tracks in the queue window.

tracks_max

The maximum number of tracks to keep open.

DLNA playlist engine plugin

Library file: **mmr-dlnaplaylist-engine.so**

This plugin supports the "`dlnatrack`" and "`dlnaplaylist`" input types. There are no parameters for this plugin.

Single-track engine plugin

Library file: **mmr-track-engine.so**

This plugin supports the "`track`" input type. There are no parameters for this plugin.

MMF audio/video player routing plugin

Library file: **mmr-mmf-routing.so**

This plugin has the following parameters:

updateinterval

The interval between status updates, in milliseconds.

BB OS audio management plugin

Library file: **mmr-audiomgmt-plugin.so**

This plugin supports the `audio:` output URL, and has these parameters:

writer

The filter to use as the audio writer for `audio:` outputs.

BB OS network connection monitoring plugin

Library file: **mmr-netmgmt-plugin.so**

There are no parameters for this plugin.

Command line for `mm-renderer`

*Start **mm-renderer** and configure context handle policies, PPS objects, and file permissions*

Synopsis:

```
mm-renderer [-cefoq] -r statepath -s serverpath
        [-U] {username|uid[:gid[,gid]*]} [-u] [-v[v...]]
```

Options:

`-c`

Destroy a context when the primary handle is closed.

`-e`

Log to *stderr* instead of slog.

`-f`

Stay in the foreground.

`-o`

Disallow the opening of existing contexts (also implies `-c`).

`-q`

Run in quiet mode.

`-r` *statepath*

The location of the PPS directory that stores the objects used by the **mm-renderer** process. We refer to this directory as the *PPS state directory*, and the default value is **/pps/services/multimedia/renderer**.

If you want to run multiple **mm-renderer** instances, you must use different PPS state directories for each instance by providing different paths with the `-r` option. Running multiple **mm-renderer** instances can improve security. For example, you could run a coporate **mm-renderer** that is accessible to only privileged system processes and a personal **mm-renderer** for use by your client applications. This way, no client could accidentally or intentionally overwrite system memory with buggy or harmful code.

-s ***serverpath***

The full path of the control object in PPS (default: the `control` object in the **/pps/services/multimedia/renderer** directory). The value for the `-s` option may be an absolute path or a relative path; for the latter case, the given path will be appended to the PPS state directory path.

-U { *username* | *uid* [:*gid* [,*gid*]*] }

Run **mm-renderer** with the given username or with the given user ID (uid) and possibly one or many group IDs (gids). When this option isn't specified, **mm-renderer** uses the client's user ID and group ID.

-u

Don't reset the umask. Without this option, the umask is reset to 0 when **mm-renderer** starts.

-v

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

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

Description:

The **mm-renderer** command line lets you adjust the context handle and logging policies, override the default PPS control object and the state directories, and assign specific file permissions to output files.

The **mm-renderer** service runs as a server process and responds to media playback commands, and delivers events to clients so they can monitor media operations.

Working with contexts

Contexts define media flows from an input to one or many outputs. You must configure a context before you can start playing media content.

To create a new context, call the *mmr_context_create()* (p. 47) function, passing in the **mm-renderer** connection handle. The creation operation returns a context handle (the *primary* handle), which you use to manipulate the context by setting parameters, attaching an input and one or more outputs, and issuing playback commands.

You can create multiple contexts, as long as your application manages potentially conflicting playback situations (e.g., simultaneous requests to play two different tracks from the same CD).

The state of a context is stored in a PPS object. For information on PPS objects, see "*PPS objects* (p. 35)".

When a context is no longer needed, you can explicitly destroy it by passing its primary handle to the *mmr_context_destroy()* (p. 49) function.

Related Links

PPS objects (p. 35)
The multimedia renderer service stores a variety of information using Persistent Publish/Subscribe (PPS) objects.

mmr_context_create() (p. 47)
Create a context

mmr_context_close() (p. 46)
Close a context

mmr_context_destroy() (p. 49)
Destroy a context

mmr_context_open() (p. 50)
Open an existing context

Closing context handles

Depending on the **mm-renderer** configuration, client applications may be allowed to obtain additional handles to existing contexts. This configuration setting determines the behavior of **mm-renderer** when closing context handles.

Handles obtained by opening existing contexts are called *secondary* context handles, whereas the handle obtained by creating a context is called the *primary* context handle.

In addition, the configuration may allow a context to exist after its primary handle has been closed. This is called an *orphan* context. When orphan contexts are allowed, secondary handles are also allowed; however, secondary handles may be allowed when orphan contexts aren't. For further clarification, see the descriptions of the `-c` and `-o` options for the ***mm-renderer*** *command-line* (p. 22).

The behavior of **mm-renderer** in closing context handles depends on your orphan context policy:

- If orphan contexts are allowed, the only way to close the primary context handle without destroying the context is by calling *mmr_context_close()* (p. 46).

- If orphan contexts aren't allowed, there's no way to close the primary handle without destroying the context. In this case, calling *mmr_context_close()* with the primary handle is equivalent to calling *mmr_context_destroy()* (p. 49) because **mm-renderer** will not only close the context handle but will also stop playback, detach any inputs and outputs, and destroy the context. Therefore, you must ensure that any secondary handles, if permitted, are properly closed.
- Regardless of the configuration, if your application terminates unexpectedly or disconnects from **mm-renderer** without explicitly closing the primary handle, the context is destroyed.

In all circumstances, it's important to properly close unneeded context handles to prevent memory leaks.

Related Links

mmr_context_close() (p. 46)
Close a context

mmr_context_destroy() (p. 49)
Destroy a context

Defining Parameters

Parameters allow you to set various properties that influence how media files are accessed and rendered during playback.

Properties such as the audio volume or video display size can be controlled by defining parameters for a context or for its input or any of its outputs. Parameters are represented as *dictionary objects* (collections of key-value pairs), where both the key and value are strings. The parameters that apply to the context and its input and outputs depend on the media content being played or recorded.

Whether defining parameters for the context, its input, or one of its outputs, your application has to call *strm_dict_new()* (p. 124) to create a new dictionary object if none exists. Use *strm_dict_set()* (p. 125) to set the key-value pairs for the parameters that you want to define.

To set parameters for the context, call the function *mmr_context_parameters()* (p. 52), passing in the handle to the dictionary object that holds the context parameters. Similarly, call *mmr_input_parameters()* (p. 89) to set input parameters and *mmr_output_parameters()* (p. 101) for output parameters. In each case, you must pass in a handle to a separate dictionary object populated with the appropriate key-value pairs.

To modify parameters, call the appropriate function again, passing in a handle to a dictionary object populated with the new parameters. Note that the *mmr_*_parameters()* functions replace any previous parameter settings with the latest settings, so the caller must keep track of which parameters have been defined. Also, the parameter functions consume the dictionary object handle in each call. If you want to keep a dictionary, call *strm_dict_clone()* (p. 114) to duplicate the handle before calling one of the parameter functions.

Related Links

Dictionary Object API (p. 113)
A dictionary object is a collection of key-value pairs that maps the names of parameters to their values. For **mm-renderer**, you can use the dictionary API to define context, input, and output parameters. Other components can use the same API to manage parameters specific to their purpose.

strm_dict_clone() (p. 114)
Duplicate a dictionary handle

strm_dict_new() (p. 124)
Create a new handle for an empty dictionary object

strm_dict_set() (p. 125)
Modify a dictionary entry (using key-value strings)

mmr_context_parameters() (p. 52)
Set context parameters

mmr_input_parameters() (p. 89)
Set input parameters

mmr_output_parameters() (p. 101)
Set output parameters

mmr_track_parameters() (p. 92)
Set track parameters

Playing media

Playing media in **mm-renderer** requires configuring a context, attaching outputs and an input, and then issuing playback commands.

To play media in **mm-renderer**:

1. Connect to **mm-renderer** using the function *mmr_connect()* (p. 41).
2. Create a new context and set the appropriate context parameters. Use the functions *mmr_context_create()* (p. 47) and *mmr_context_parameters()* (p. 52).
3. Attach an output and set its output parameters. Use the functions *mmr_output_attach()* (p. 96) and *mmr_output_parameters()* (p. 101). You can attach multiple outputs.
4. Attach the input and set the input parameters. Use the functions *mmr_input_attach()* (p. 84) and *mmr_input_parameters()* (p. 89).
5. Start playback for the context by calling *mmr_play()* (p. 108).

The media starts to play. You can stop playback by calling *mmr_stop()* (p. 112).

Related Links

Multimedia Renderer API (p. 39)
The multimedia renderer API exposes the data types and functions you can use to connect to **mm-renderer**, create contexts, configure inputs and outputs, control playback, and process events.

Play states

The possible play states of the context are:

Idle

No input is attached.

Stopped

Input is attached but is not playing.

Playing

Input is attached and is playing.

Note that there is no Paused play state. Paused playback is represented by a play speed of `0`.

Play speed

In **mm-renderer**, the play speed is represented by an integer. Normal speed is represented by a value of `1000`, and `0` means paused. Depending on the context's input media, trick play speeds (i.e., negative, slower than normal, or faster than normal) may not be supported.

Use the *mmr_speed_set()* (p. 110) function to change the current play speed. You can change the speed when the state is *stopped*; **mm-renderer** simply saves the setting and applies it when playback restarts.

Seeking to positions

Use the *mmr_seek()* (p. 109) function to seek to a known position in a single track or a track within a playlist. If the current context input is a track, simply specify the track position in milliseconds, for example `"2500"`. If the context input is a playlist, the position must be a string in the format `"99:9999"`, for example `"2:1200"`, where the first number is the track position in the current playlist and the second number is the number of milliseconds from the beginning of the specified track.

Related Links

mmr_seek() (p. 109)
Seek to a position

Managing video windows

You can render video to a display using the Screen Graphics Subsystem library.

The following example shows how to give **mm-renderer** a window group and window ID to use in creating a window on the application's behalf, configure **mm-renderer** for audio and video output, and get a handle to the window and use the Screen API functions to manipulate the output.

To begin, we define a window name to use as the window ID and retrieve the unique group name created by *screen_create_window_group()*. We use these two properties to set the output URL, *video_device_url*.

```
const char *window_name = "appwindow";
char *window_group_name;
int MAX_WINGRP_NAME_LEN = 49;
window_group_name = (char *)malloc(MAX_WINGRP_NAME_LEN);

// Create the video URL for mm-renderer
static char video_device_url[PATH_MAX];

// Create a window group.
// Pass NULL to generate a unique window group name.
if (screen_create_window_group(g_screen_win, NULL) != 0) {
    return EXIT_FAILURE:
}

// Get the window group name.
rc = screen_get_window_property_cv( g_screen_win,
                                    SCREEN_PROPERTY_GROUP,
                                    PATH_MAX,
                                    window_group_name );
if (rc != 0) {
    fprintf( stderr,
     "screen_get_window_property(SCREEN_PROPERTY_GROUP) failed.\n" );
    return EXIT_FAILURE;
}
rc = snprintf( video_device_url,
```

```
                PATH_MAX,
                "screen:?winid=%s&wingrp=%s",
                window_name,
                window_group_name );

if (rc < 0) {
    fprintf(stderr, "Error building video device URL string\n");
    return EXIT_FAILURE;
}
else if (rc >= PATH_MAX) {
    fprintf(stderr, "Video device URL too long\n");
    return EXIT_FAILURE;
}

// Create the video context name for mm-renderer
static const char *video_context_name = "videoContext";
```

After the window group is created, we connect to **mm-renderer** and create a context. We then attach the video output to the context by calling *mmr_output_attach()* (p. 96), specifying the URL variable that we set up earlier. We use the same function to attach the audio output.

```
// Configure mm-renderer
mmr_connection = mmr_connect(NULL);
if (mmr_connection == NULL) {
    fprintf(stderr, "Error connecting to renderer service: %s\n",
            strerror(errno));
    return EXIT_FAILURE;
}

mmr_context = mmr_context_create( mmr_connection,
                                  video_context_name,
                                  0,
                                  S_IRWXU|S_IRWXG|S_IRWXO );
if (mmr_context == NULL) {
    fprintf(stderr, "Error creating renderer context: %s\n",
            strerror(errno));
    return EXIT_FAILURE;
}

// Configure video and audio output
const mmr_error_info_t* errorInfo;
video_device_output_id = mmr_output_attach( mmr_context,
                                            video_device_url,
                                            "video" );
if (video_device_output_id == -1) {
    errorInfo = mmr_error_info(mmr_context);
    fprintf(stderr, "Attaching video output produced error code \
```

```
            %d\n", errorInfo->error_code);
    return EXIT_FAILURE;
}

audio_device_output_id = mmr_output_attach( mmr_context,
                                            audio_device_url,
                                            "audio" );
if (audio_device_output_id == -1) {
    // Call mmr_error_info(), display an error message, and exit
    ...
}
```

Next, we retrieve the handle of the video window from the screen event received when the window is created, and check that the ID of the window indicated in the event matches our output video window. For more complicated applications, this is important so that we can distinguish between our video window and another child window belonging to the same window group.

All functions used here are from the Screen API.

```
// Create the screen context, which is needed to retrieve the event
screen_context_t screen_ctx = 0;
if ( screen_create_context( &screen_ctx,
                            SCREEN_APPLICATION_CONTEXT) != 0 ) {
    fprintf(stderr, "Error creating screen context: %s\n",
            strerror(errno));
    return EXIT_FAILURE;
}

screen_event_t screen_event;
screen_create_event(&screen_event);

// Set a timeout of -1 to block until an event is received
screen_get_event(screen_ctx, screen_event, -1);
int event_type;
screen_get_event_property_iv( screen_event,
                              SCREEN_PROPERTY_TYPE,
                              &event_type );

// Check if it's a creation event and the video output window
// has not yet been initialized
if ((event_type == SCREEN_EVENT_CREATE) &&
        (video_window == (screen_window_t)NULL)) {
    char id[256];

    rc = screen_get_event_property_pv( screen_event,
                                       SCREEN_PROPERTY_WINDOW,
                                       (void**)&video_window );
    if (rc != 0) {
```

```
        fprintf(stderr, "Error reading event window: %s\n",
                strerror(errno));
        return EXIT_FAILURE;
    }

    rc = screen_get_window_property_cv( video_window,
                                        SCREEN_PROPERTY_ID_STRING,
                                        256,
                                        id );
    if (rc != 0) {
        fprintf(stderr, "Error reading window ID: %s\n",
            strerror(errno));
        return EXIT_FAILURE;
    }

    if (strncmp(
            id, window_group_name, strlen(window_group_name)) != 0)
        fprintf(stderr, "Mismatch in window group names\n");
        return EXIT_FAILURE;
}
```

After we have this handle, we can manipulate the video window directly with Screen API calls.

```
// Set the z-order of the video window to put it above or below
// the main window. Alternate between +1 and -1 to implement
// double-buffering to avoid flickering of output.
app_window_above = !app_window_above;
if (app_window_above) {
    screen_val = 1;
}
else {
    screen_val = -1;
}

if (screen_set_window_property_iv( video_window,
                                   SCREEN_PROPERTY_ZORDER,
                                   &screen_val ) != 0) {
    fprintf(stderr, "Error setting z-order of video window: %s\n",
            strerror(errno));
    return EXIT_FAILURE;
}

// Set the video window to be visible.
screen_val = 1;
if (screen_set_window_property_iv( video_window,
                                   SCREEN_PROPERTY_VISIBLE,
                                   &screen_val) != 0 ) {
```

```
        fprintf(stderr, "Error making window visible: %s\n",
                strerror(errno));
        return EXIT_FAILURE;
    }
    ...
```

Related Links

mmr_connect() (p. 41)
Connect to ***mm-renderer***

mmr_context_create() (p. 47)
Create a context

mmr_output_attach() (p. 96)
Attach an output

mmr_error_info() (p. 60)
Get error information

Windows functionality in Screen

Contexts functionality in Screen

Events functionality in Screen

Recording audio data

You can record audio content in **mm-renderer** by attaching the input to an audio capture device and directing the output to a file instead of a device.

The following sample program shows how to give **mm-renderer** an input URL of type `snd:` to select and configure an audio capture device (microphone), set an output URL type of `file:` to target a file, and then start and stop playback to record captured audio content to the targetted file. The `snd:` input URL format works only with the `file:` output type, so your code must obey this design.

You can record audio content for as long as you like, but you must ensure your client application's output file can hold all the content you want to capture. The size of the generated output depends on many settings, including the sampling rate and number of channels. This sample program records in mono by specifying one channel (*nchan=1*) in the input URL. Depending on your platform, your microphone device might have two recorders, so you could record in stereo by setting two channels (*nchan=2*). You could also increase the sampling rate to attain the necessary audio quality, such as using the standard CD sampling rate of 44.1 MHz (*frate=44100000*). For more information on the available device options, see the *list of URL parameters for audio capture devices* (p. 86).

This code sample names an AMR file for the output, but **mm-renderer** supports other formats, such as wideband AMR (see the *list of supported output file formats* (p. 97)).

```
void record_AMR_file()
{
  mmr_connection_t *connection;
  mmr_context_t *context;
  const char* context_name = "AnyNameYouWant";
  int output = 0;
  const char* outputFile = "/tmp/testFile.amr";
  int input = 0;

  connection = mmr_connect(NULL);

  if (connection) {
      context = mmr_context_create( connection,
                                    context_name,
                                    0,
                                    S_IRWXU );

      if (context) {
          // specify a file output so the audio content is
          // not played but recorded in a file
          output = mmr_output_attach( context,
                                      outputFile,
                                      "file" );

          // specify the audio device under /dev/snd you want to
          // use for the recording, and the recording details
```

```
                // (in this case, we use a sampling rate of 8000 Hz and
                // 1 channel for mono (not stereo) recording)
                input = mmr_input_attach( context,
                        "snd:/dev/snd/pcmPreferredc?nchan=1&frate=8000",
                                        "track" );

                // start recording
                mmr_play(context);

                // delay for the length of time you want to record
                // (in this case, 30 seconds)
                sleep(30);

                // stop recording
                mmr_stop(context);

                // clean up the context
                mmr_input_detach(context);
                mmr_output_detach(context, output);
                mmr_context_destroy(context);
            }
            mmr_disconnect(connection);
        } // if (connection)
    } // function
```

Related Links

mmr_connect() (p. 41)
*Connect to **mm-renderer***

mmr_context_create() (p. 47)
Create a context

mmr_output_attach() (p. 96)
Attach an output

mmr_error_info() (p. 60)
Get error information

mmr_input_attach() (p. 84)
Attach an input

mmr_play() (p. 108)
Start playing

mmr_stop() (p. 112)
Stop playing

PPS objects

The multimedia renderer service stores a variety of information using Persistent Publish/Subscribe (PPS) objects.

PPS objects are implemented as files in a special filesystem. The PPS objects created by **mm-renderer** are located in subdirectories under the PPS root directory (**/pps/services/multimedia/renderer**).

These objects store information about:

- context state
- play state, warnings, and errors
- input metadata
- playlists
- supported file types

To get information from PPS objects, you can use the POSIX *open()* and *read()* functions, or you can use functions from the PPS encoding and decoding API, which is explained in the *PPS Developer's Guide*.

The attributes of some objects (e.g., the `status` object) might get refreshed very frequently, so you shouldn't use delta mode to read these objects. For more information about delta mode, see the "Subscription Modes" section in the *PPS Developer's Guide*.

Context state

Every time you create a context by calling *mmr_context_create()*, **mm-renderer** also creates a context directory with the same name. For instance, creating a context named **movie1** creates a directory named **/pps/services/multimedia/renderer/context/movie1**.

Inside each context directory, **mm-renderer** creates several objects (files) that hold the state of the context. When an input is attached to the context, additional objects may be created in the context directory, depending on the input type.

The state of a context is represented by the following objects, where the # character indicates a numeric value encoded as decimal:

`param`

Contains the parameters set with *mmr_context_parameters()*.

`output#`

Created when an output is attached, deleted when it's detached. The # token is the output ID returned by *mmr_output_attach()*. This object contains the URL, output type, and the latest parameters set with *mmr_output_parameters()*.

`input`

Populated when an input is attached, emptied when it's detached. This object contains the URL, input type, and the latest parameters set with *mmr_input_parameters()*.

status

A snapshot of the current status. This is potentially high bandwidth, so delta mode shouldn't be used to read this object. This object holds information on the playback position, the buffer capacity, and the buffer activity.

state

The play state. This object is intended to be read in delta mode, otherwise errors or warnings may be lost. Depending on the play state, the `state` object may have these attributes:

state

`playing`, `stopped`, or `idle`.

speed

The current speed, in units of 1/1000th of normal speed.

warning

The most recent warning (which is deleted when playback is stopped).

warning_pos

The play position when the warning happened.

error

The most recent error code (which is deleted when playback is restarted).

error_pos

The play position when the error happened.

Play state, warnings, and errors

To detect changes in the play state, read the `state` object in delta mode.

The `state` object is updated based on input attachment and playback events, as follows:

- When there's no input attached, the `state` attribute (within the `state` object) is `idle` and no other attributes are present.
- When an input is attached, the `state` attribute changes from `idle` to `stopped`.
- When playback begins, the `state` attribute changes from `stopped` to `playing` and any `error` and `error_pos` attributes are deleted.
- When the end of media is reached, the `state` attribute changes from `playing` to `stopped` and the `error` attribute is set to MMR_ERROR_NONE.
- When playback is stopped by a function call, no error code is published to the `state` object.

A warning is a problem that doesn't stop playback. If there's a warning, the `state` remains as `playing` and the `warning` and `warning_pos` attributes are set.

An error is a problem that stops playback. If there's an error:

- the `warning` and `warning_pos` attributes, if any, are deleted
- the `state` attribute is set to `stopped`
- the `error` and `error_pos` attributes are set

Input metadata

To get metadata for the main input, read the `metadata` object, which **mm-renderer** creates in the corresponding PPS context directory when an input is attached to the context.

When the main input is a track played independently or a playlist, the `metadata` object stores attributes that correspond to the main input's metadata fields. For example, the `metadata` object for an audio track has attributes such as `md_title_album`, `md_title_artist`, and `md_title_bitrate`.

When the main input is an *autolist*, which is a single track formatted as a playlist, the `metadata` object stores only the URL of the track.

To get metadata for media tracks that are playlist or autolist entries, read the `q#` objects, which are stored in the same PPS context directory. There is one `q#` object for each playlist entry, where the `#` token is the position of the track in the playlist (starting from 1). For autolists, there is only the `q1` object. The metadata attributes for a track are set when the track begins to play.

Playlist window

When the context input is a playlist, **mm-renderer** creates additional PPS objects in the context directory. These PPS objects specify the currently playing item and the items in front of and behind the current item, up to a preconfigured maximum; this information is collectively known as the *playlist window*.

The following PPS objects represent the playlist window:

`p#`

Contains the track parameters for a playlist entry. There is one `p#` object for each playlist entry, where the `#` token is the position of the track in the playlist (starting from 1). Each such object contains the latest parameters passed into *mmr_track_parameters()* for the corresponding track.

`play-queue`

Represents the size of the playlist window. The `play-queue` object has the following attributes:

`start`

The index of the first `p#` item in the window.

`end`

The index of the last `p#` item in the window.

`total`

The total number of items in the playlist; this is set whenever the full length of the playlist is known.

If you seek to a track outside of the playlist window, the indexes of the first and last items may retain stale values for a short time after the seek command is issued. This is because these attributes are updated asynchronously.

Supported file and MIME types

The **/pps/services/multimedia/renderer/component** directory contains the **.all** object, which lists the supported file extensions and MIME types. To access this information, examine the following attributes in the **.all** object:

audioencodeextensions

Lists supported filename extensions for file outputs, in a comma-separated list (e.g., `m4a,wav`).

mime

Lists allowed combinations of playable MIME types, in a comma-separated list (e.g., `3gpp,video`).

Applications should be prepared to merge value sets listed in multiple instances of the same attribute.

Chapter 3
Multimedia Renderer API

The multimedia renderer API exposes the data types and functions you can use to connect to **mm-renderer**, create contexts, configure inputs and outputs, control playback, and process events.

To play media with **mm-renderer**, your client application must first connect to the service. Next, it can create a context to manage the media flow from an input to one or more outputs. It can also specify parameters for the context, its input, and its outputs. After the parameters have been configured, your client can issue media playback commands; for example, to start and stop playback.

The API can deliver events, which indicate playback state changes, new input and output attachments, parameter updates, or errors. An application can read information on the latest event from a special-purpose data type and then perform the necessary processing.

The header file that defines most of the API functions, **renderer.h**, is located in **${*QNX_TARGET*}/usr/include/mm/** on the development system (not the target). The enumerated error codes and most of the API data types are defined in **types.h**, which is located in **${*QNX_TARGET*}/usr/include/mm/renderer/**. This same subdirectory stores **events.h**, which defines the event-processing functions and data types.

Connection management

You must connect to **mm-renderer** before you can use it to define contexts, configure inputs and outputs, and issue playback commands.

The *mmr_connect()* function returns a valid connection handle, when successful. This handle must be passed in to the subsequent API calls for creating or opening a context. When the context is created or opened, it returns another handle that you must use for all media operations related to that context.

The connection handle isn't needed again until you're finished with the connection. At this point, you must close the connection by calling *mmr_disconnect()*.

mmr_connect()

*Connect to **mm-renderer***

Synopsis:

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

mmr_connection_t* mmr_connect( const char *name )
```

Arguments:

name

The name of the **mm-renderer** service to connect to (use NULL for the default service).

Library:

mmrndclient

Description:

Connect to **mm-renderer**, using the specified name if *name* isn't `NULL`, returning a valid connection handle on success.

Returns:

A connection handle, or NULL on failure (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_connection_t

*The **mm-renderer** connection handle type*

Synopsis:

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

typedef struct mmr_connection mmr_connection_t;
```

Library:

mmrndclient

Description:

The **mmr_connection_t** structure is a private data type representing the connection to **mm-renderer**.

Classification:

QNX Neutrino

mmr_disconnect()

*Disconnect from **mm-renderer***

Synopsis:

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

void mmr_disconnect( mmr_connection_t *connection )
```

Arguments:

connection

An **mm-renderer** connection handle.

Library:

mmrndclient

Description:

Disconnect from **mm-renderer**. Close any existing context handles associated with the connection being closed and free their memory. You shouldn't use these handles again, not even in an API call to close them. If any of them are primary handles, their contexts also get destroyed.

The same happens in terms of contexts being destroyed if your application exits without explicitly disconnecting. This means you don't have to clean up old contexts when you restart the application.

Each context handle is associated with the connection handle used to create it. This means that if you have multiple connections to **mm-renderer**, calling *mmr_disconnect()* to close one of those connections doesn't necessarily close all your context handles.

This function is asynchronous and may return before the destruction of any related contexts is complete. Calling *mmr_context_destroy()* (p. 49) for each context associated with the connection ensures that the context is destroyed before returning.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Context management

Each **mm-renderer** context manages a separate media flow independently of other contexts. The media flow is determined by the input and the outputs that you attach to the context as well as the input and output parameters. You can also set context parameters, which are independent of the media files being played.

To create a context, call *mmr_context_create()*. Depending on the **mm-renderer** configuration, you may be able to open an existing context by calling *mmr_context_open()*. Both of these functions return a context handle that you must pass in to subsequent API calls to perform media operations on the same context.

The sequence of API calls needed to access a context, set its parameters, define an input and output (and set their parameters), and start playback is outlined in "*Playing media* (p. 27)".

When you're finished using a context, you must close it to properly free resources. You may want to or have to destroy the context in addition to closing it, based on your application needs and **mm-renderer** configuration. The behavior of the **mm-renderer** service when closing context handles is complex, as explained in detail in "*Working with contexts* (p. 24)".

mmr_command_send()

Send a remote control command to the context

Synopsis:

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

int mmr_command_send( mmr_context_t *ctxt, const char *cmd )
```

Arguments:

ctxt

A context handle.

cmd

The command to send.

Library:

mmrndclient

Description:

Send a remote control command to the context. The commands available depend on the plugin in use.

This function is offered for future use; currently, no commands are defined.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_context_close()

Close a context

Synopsis:

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

int mmr_context_close( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Close and invalidate the context handle. The handle passed to *mmr_context_close()* always gets closed and becomes invalid, even if the function returns an error. If the primary handle (which was returned by *mmr_context_create()*) is passed in, the associated context might be destroyed, depending on the configuration. If this is the case, the function fails and sets the global variable *errno* to `EPERM`.

Returns:

Zero on success, -1 on failure (check *errno*). The handle becomes invalid either way.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Working with contexts (p. 24)
Contexts define media flows from an input to one or many outputs. You must configure a context before you can start playing media content.

mmr_context_create()

Create a context

Synopsis:

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

mmr_context_t* mmr_context_create( mmr_connection_t *connection,
                                   const char *name,
                                   unsigned flags,
                                   mode_t mode )
```

Arguments:

connection

An **mm-renderer** connection handle.

name

The name of the context. This must be a valid filename and will show up in the pathname space as a directory.

flags

Must be zero. No flags are defined for now.

mode

Permission flags controlling which processes can access the context. These flags are specified in a standard POSIX permissions bitfield.

The `w` bits control which processes can open secondary handles to access the context. The `r` and `x` bits provide access to **mm-renderer** events related to the context.

In this bitfield, the user permissions apply to the caller and to any process with the same effective user ID (`euid`). You must set these permissions appropriately to grant your application (or other applications running with the same `euid`) sufficient access to the context being created. The group permissions apply to processes with an effective group ID (`egid`) or a supplementary group ID matching the caller's `egid`. The other permissions apply to all other processes.

Library:

mmrndclient

Description:

Create and open a new context with the specified name. Fail if a context with that name already exists. The name must be a valid filename and will show up as a directory in the pathname space, with its file permissions set based on the *mode* argument. Note that there's not a direct mapping between the

value given in *mode* and the file permissions assigned to the context directory. For an explanation of how the permissions specified in the function call are interpreted, see the *mode* (p. 47) argument.

When successful, the function returns a handle, called the *primary* handle, for accessing the newly created context. Depending on your configuration, you may be able to create any number of secondary handles by calling *mmr_context_open()* (p. 50).

To avoid memory leaks, every handle opened with *mmr_context_create()* needs to be closed, either explicitly through an API call or implicitly by terminating the process. The **mm-renderer** configuration also determines whether closing the primary handle also destroys the context. If this option is set and you do close the primary handle of a context, you can no longer use any secondary handles to that context, so you must close those handles by calling *mmr_context_close()* (p. 46) on each one. If this option isn't set, you can call *mmr_context_close()* to close the primary handle without destroying the context, which lets you keep using that context by accessing it with secondary handles.

Returns:

A handle on success, or a null pointer on failure (check *errno*).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Working with contexts (p. 24)
Contexts define media flows from an input to one or many outputs. You must configure a context before you can start playing media content.

mmr_context_destroy()

Destroy a context

Synopsis:

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

int mmr_context_destroy( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Destroy the context the handle refers to and close the handle. Implicitly stop any playback and detach any input or outputs. If any other handles to this context still exist, attempts to use them will fail. At this point, you should close those handles by calling *mmr_context_close()* on each one.

In addition to calling *mmr_context_destroy()*, you can destroy a context in these ways:

- calling *mmr_context_close()* using the primary handle (if your configuration disallows orphan handles)
- calling *mmr_disconnect()* (if you have the primary handle)
- terminating the process (if you have the primary handle)

Returns:

Zero on success, -1 on failure (check *errno*). The handle becomes invalid either way.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Working with contexts (p. 24)
Contexts define media flows from an input to one or many outputs. You must configure a context before you can start playing media content.

mmr_context_open()

Open an existing context

Synopsis:

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

mmr_context_t* mmr_context_open( mmr_connection_t *connection,
                                 const char *name )
```

Arguments:

connection

An **mm-renderer** connection handle.

name

The context name.

Library:

mmrndclient

Description:

Open a handle to an existing context. The handle returned by this function is called a *secondary* handle.

Whether this operation is allowed depends on the options that you define for the **mm-renderer** process (for details, see "*mm-renderer command line* (p. 22)"). If you set these options to disallow the opening of secondary handles, this function fails and sets the global variable *errno* to `EPERM`. If these options allow the opening of secondary handles, you can open as many as you like, and the function will return a new handle with each successful call.

To avoid memory leaks, every handle opened with *mmr_context_open()* needs to be closed, either explicitly through an API call or implicitly by terminating the process.

Returns:

A handle on success, or a null pointer on failure (check *errno*).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Working with contexts (p. 24)
Contexts define media flows from an input to one or many outputs. You must configure a context before you can start playing media content.

mmr_context_parameters()

Set context parameters

Synopsis:

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

int mmr_context_parameters( mmr_context_t *ctxt, strm_dict_t *parms )
```

Arguments:

ctxt

A context handle.

parms

A reference to a dictionary containing the context parameters to set (must not be NULL). Any previous parameters are overridden.

The **strm_dict_t** object becomes API property after this call, even if the call fails. You should not use or destroy the dictionary after passing it to this function.

Library:

mmrndclient

Description:

Set parameters associated with the specified context. The applicable parameters and their types and values are implementation-specific. For example, different input and output types may require different parameters associated with the context. In general, the following parameter is supported:

updateinterval

Allows an application to request a particular frequency in status updates from **mm-renderer**. How accurately this delivery reflects the *updateinterval* setting depends on the plugin handling the media flow. Currently, this parameter is supported only for the MMF audio/video player routing plugin. The default update interval is 1000 ms, but your client code should dynamically adjust this parameter based on the application's state, such as fullscreen versus minimized versus when the screen is off. You can also override this parameter in the configuration file.

QNX Neutrino RTOS supports the following parameters that map to **libcurl** options:

- *OPT_VERBOSE*
- *OPT_CONNECTTIMEOUT_MS*
- *OPT_LOW_SPEED_LIMIT*
- *OPT_LOW_SPEED_TIME*
- *OPT_USERAGENT*
- *OPT_USERNAME*

- *OPT_PASSWORD*
- *OPT_PROXYUSERNAME*
- *OPT_PROXYPASSWORD*
- *OPT_COOKIE*
- *OPT_COOKIEFILE*
- *OPT_COOKIEJAR*
- *OPT_COOKIESESSION*
- *OPT_CAINFO*
- *OPT_CAPATH*
- *OPT_SSL_VERIFYPEER*
- *OPT_SSL_VERIFYHOST*
- *OPT_PROXY*
- *OPT_NOPROXY*
- *OPT_HTTPPROXYTUNNEL*
- *OPT_PROXYPORT*
- *OPT_PROXYTYPE*
- *OPT_PROXYAUTH*
- *OPT_HTTPAUTH*
- *OPT_HTTPHEADER*
- *OPT_DNSCACHETIMEOUT*

QNX Neutrino RTOS supports the following parameters that map to socket options (see the *getsockopt()* function in the *C Library Reference* for more information):

- *OPT_SO_RCVBUF*
- *OPT_SO_SNDBUF*

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Defining Parameters (p. 26)
Parameters allow you to set various properties that influence how media files are accessed and rendered during playback.

strm_dict_t (p. 131)
Dictionary object type

mmr_context_t

*The **mm-renderer** context handle type*

Synopsis:

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

typedef struct mmr_context mmr_context_t;
```

Library:

mmrndclient

Description:

The **mmr_context_t** structure is a private data type representing a context handle. Your application can monitor changes to the context state by using the *event API functions* (p. 63).

Classification:

QNX Neutrino

Error information

The **mm-renderer** service stores information about any error that occurs when an API call is made. When requested, this error information is returned in a structure that includes a code for the **mm-renderer** error type and other information specific to either the media protocol used or to the underlying POSIX error.

It's good practice to check for errors after each API call, to promptly learn of any problems with the media configuration or playback. You can do this by examining each function call's return code and if it indicates failure (e.g., by a non-zero value, often -1), calling *mmr_error_info()* to get information about the error that occurred. The action to perform following a given error type is a design choice.

mm_error_code_t

*Error codes set by **mm-renderer** API functions*

Synopsis:

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

typedef enum mm_error_code {
    MMR_ERROR_NONE,
    MMR_ERROR_UNKNOWN,
    MMR_ERROR_INVALID_PARAMETER,
    MMR_ERROR_INVALID_STATE,
    MMR_ERROR_UNSUPPORTED_VALUE,
    MMR_ERROR_UNSUPPORTED_MEDIA_TYPE,
    MMR_ERROR_MEDIA_PROTECTED,
    MMR_ERROR_UNSUPPORTED_OPERATION,
    MMR_ERROR_READ,
    MMR_ERROR_WRITE,
    MMR_ERROR_MEDIA_UNAVAILABLE,
    MMR_ERROR_MEDIA_CORRUPTED,
    MMR_ERROR_OUTPUT_UNAVAILABLE,
    MMR_ERROR_NO_MEMORY,
    MMR_ERROR_RESOURCE_UNAVAILABLE,
    MMR_ERROR_MEDIA_DRM_NO_RIGHTS,
    MMR_ERROR_DRM_CORRUPTED_DATA_STORE,
    MMR_ERROR_DRM_OUTPUT_PROTECTION,
    MMR_ERROR_DRM_OPL_HDMI,
    MMR_ERROR_DRM_OPL_DISPLAYPORT,
    MMR_ERROR_DRM_OPL_DVI,
    MMR_ERROR_DRM_OPL_ANALOG_VIDEO,
    MMR_ERROR_DRM_OPL_ANALOG_AUDIO,
    MMR_ERROR_DRM_OPL_TOSLINK,
    MMR_ERROR_DRM_OPL_SPDIF,
    MMR_ERROR_DRM_OPL_BLUETOOTH,
    MMR_ERROR_DRM_OPL_WIRELESSHD,
    MMR_ERROR_DRM_OPL_RESERVED_LAST =
            MMR_ERROR_DRM_OPL_WIRELESSHD + 4,
    MMR_ERROR_MEDIA_DRM_EXPIRED_LICENSE,
    MMR_ERROR_PERMISSION,
    MMR_ERROR_COUNT,
} mm_error_code_t;
```

Data:

MMR_ERROR_NONE

No error has occurred. This error code is used for the EOF event but never returned as the error code from an API call.

MMR_ERROR_UNKNOWN

An unexpected error.

MMR_ERROR_INVALID_PARAMETER

An invalid parameter, such as an invalid output ID or a seek string that's incorrectly formatted or out of range.

MMR_ERROR_INVALID_STATE

An illegal operation given the context state, such as attempt to play or seek when no input is attached, to change the playlist when playback was stopped, or to access the context after it's been destroyed.

MMR_ERROR_UNSUPPORTED_VALUE

An unrecognized input or output type or an out-of-range speed setting.

MMR_ERROR_UNSUPPORTED_MEDIA_TYPE

An unrecognized data format.

MMR_ERROR_MEDIA_PROTECTED

The file is DRM-protected and either it uses an unsupported DRM scheme or there's a DRM error not corresponding to any of the errors listed below.

MMR_ERROR_UNSUPPORTED_OPERATION

An illegal operation. This error is returned if you try to seek or to set the playback speed on media that doesn't allow it, or you try to attach an output after attaching the input but the underlying media doesn't support that action sequence.

MMR_ERROR_READ

An I/O error at the source.

MMR_ERROR_WRITE

An I/O error at the sink.

MMR_ERROR_MEDIA_UNAVAILABLE

mm-renderer can't open the source.

MMR_ERROR_MEDIA_CORRUPTED

mm-renderer found corrupt data on the media.

MMR_ERROR_OUTPUT_UNAVAILABLE

mm-renderer can't write to the output (possibly because the output URL or type doesn't match any supported sink).

MMR_ERROR_NO_MEMORY

Insufficient memory to perform the requested operation.

MMR_ERROR_RESOURCE_UNAVAILABLE

A required resource such as an encoder or an output feed is presently unavailable.

MMR_ERROR_MEDIA_DRM_NO_RIGHTS

The client lacks the rights to play the file.

MMR_ERROR_DRM_CORRUPTED_DATA_STORE

The DRM data store is corrupted.

MMR_ERROR_DRM_OUTPUT_PROTECTION

A DRM output protection mismatch on an unspecified output.

MMR_ERROR_DRM_OPL_HDMI

A DRM output protection mismatch on an HDMI output.

MMR_ERROR_DRM_OPL_DISPLAYPORT

A DRM output protection mismatch on a DISPLAYPORT output.

MMR_ERROR_DRM_OPL_DVI

A DRM output protection mismatch on a DVI output.

MMR_ERROR_DRM_OPL_ANALOG_VIDEO

A DRM output protection mismatch on a video ANALOG output (e.g., S-VIDEO, COMPOSITE, RGB, RGBHW, YPbPr).

MMR_ERROR_DRM_OPL_ANALOG_AUDIO

A DRM output protection mismatch on an audio ANALOG output (e.g., HEADPHONE, SPEAKER OUT).

MMR_ERROR_DRM_OPL_TOSLINK

A DRM output protection mismatch on a TOSLINK output.

MMR_ERROR_DRM_OPL_SPDIF

A DRM output protection mismatch on an S/PDIF output.

MMR_ERROR_DRM_OPL_BLUETOOTH

A DRM output protection mismatch on a BLUETOOTH output.

MMR_ERROR_DRM_OPL_WIRELESSHD

A DRM output protection mismatch on a WIRELESSHD output.

MMR_ERROR_DRM_OPL_RESERVED_LAST

Identifier marking the end-of-range for MMR_ERROR_DRM_OPL_* values.

MMR_ERROR_MEDIA_DRM_EXPIRED_LICENSE

A license for the DRM file was found but has expired, either because the play count has been depleted or the end time has passed.

MMR_ERROR_PERMISSION

A playback permission error (e.g., user prohibition, region mismatch).

MMR_ERROR_COUNT

An end-of-list identifier. Also indicates the number of distinct error codes.

Library:

mmrndclient

Classification:

QNX Neutrino

mmr_error_info()

Get error information

Synopsis:

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

const mmr_error_info_t* mmr_error_info( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

This function gets error information. It returns a pointer to an internal buffer containing any error information related to the most recent API call made using the same context handle. The pointer and the error information it points to are valid only until another API call is made.

Returns:

A pointer to the error information, or a null pointer if the most recent API call succeeded.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_error_info_t

Error information for ***mm-renderer***

Synopsis:

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

typedef struct mmr_error_info {
    uint32_t error_code;
    char extra_type[ 20 ];
    int64_t extra_value;
    char extra_text[ 256 ];
} mmr_error_info_t;
```

Data:

uint32_t error_code

One of the *mm_error_code_t* (p. 56) constants.

char extra_type[20]

A short string identifying the API or protocol that defines the meaning of *extra_value*, such as "errno", "http", or "mmf".

int64_t extra_value

An error number according to *extra_type*.

char extra_text[256]

Free-form text describing the error. This may or may not have a format formally defined by a specification. For example, when *extra_type* is "http", this field contains an HTTP server response string.

Library:

mmrndclient

Description:

The **mmr_error_info_t** structure contains error information generated by **mm-renderer** functions. Use the function *mmr_error_info()* to retrieve error information for a particular context and function call.

This multifield structure allows plugins to return protocol- or API-specific error information in addition to the MMR error code. The *extra_type* string is a tag that specifies how to interpret the *extra_value* and *extra_text* fields.

The values of *extra_type* currently supported and the associated contents of the other two fields are as follows:

extra_type	extra_value	extra_text
"" (empty)	0	Usually empty, possibly some descriptive text
"errno"	An *errno* value	Usually the result of `strerror(extra_value)`, but possibly something more descriptive
"mmf"	One of the MMF-specific error codes (not a valid *errno*)	Usually empty, but possibly something more descriptive
"http"	An HTTP response code	An HTTP server response
"libcurl"	A **libcurl** error code (but not CURLE_HTTP_RETURNED_ERROR)	The corresponding **libcurl** error message

Classification:

QNX Neutrino

Events

The **mm-renderer** service can report events so you can monitor and react to changes in playback activity, input and output attachments to a context, and any errors that might occur.

The event-processing support includes functions for arming the notification of an **mm-renderer** event, waiting until an event occurs, and retrieving information on the latest event. The event information is stored in the **mmr_event_t** data type and includes but is not limited to:

- the event type
- the new context state
- the current playback speed
- a warning string, when applicable
- detailed error information, when applicable

Using this API functionality, you can process events in an automated way without having to access and parse status files to keep up with the state of the **mm-renderer** service. You can instead write an event-processing loop to continuously monitor and react to events by storing the necessary data and performing follow-up actions.

mmr_event_arm()

Set a **sigevent** *to deliver when a new event becomes available*

Synopsis:

```
#include <mm/renderer/events.h>

int mmr_event_arm( mmr_context_t *ctxt, struct sigevent const *sev )
```

Arguments:

ctxt

A context handle.

sev

A **sigevent** to send; set to NULL to disarm.

Library:

mmrndclient

Description:

Set a **sigevent** to deliver when a new **mm-renderer** event becomes available. The *mmr_event_arm()* function is helpful if your program already has an event-processing loop that uses signals or pulses as notifications and you simply want to add code that processes **mm-renderer** events. To do this, you must first call *mmr_event_arm()* to request notification of the next **mm-renderer** event. Then, in the new event-handling code, you must call *mmr_event_get()* (p. 68) to retrieve the event information.

Because *mmr_event_arm()* enables notification of only one event, you must call *mmr_event_arm()* repeatedly if you want to receive a **sigevent** for each **mm-renderer** event that occurs. The function is non-blocking because although it enables notification of an event, it doesn't wait for the event to occur.

If **mm-renderer** already has an event waiting when you call *mmr_event_arm()*, the function doesn't arm a **sigevent** but immediately returns a value greater than zero. If you receive such a value, you must call *mmr_event_get()* and process the event.

Occasionally, the *mmr_event_get()* function can't retrieve any meaningful event data and instead returns the MMR_EVENT_NONE event. This can happen if the **sigevent** wasn't armed (because an event was already waiting) or if the **sigevent** was armed and then delivered by the system (because an event occurred after the last *mmr_event_arm()* call). For an example of a situation when MMR_EVENT_NONE might be returned, see *mmr_event_wait()* (p. 80).

Returns:

A positive number if the **sigevent** isn't armed, zero on success, or -1 on failure (check *errno*).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	**No**, except when using different context handles

Related Links

mmr_event_t (p. 70)
*Information about an **mm-renderer** event*

mmr_event_data_set()

Set user data for the dictionary returned with the last event

Synopsis:

```
#include <mm/renderer/events.h>

int mmr_event_data_set( mmr_context_t *ctxt, void *usrdata )
```

Arguments:

ctxt

A context handle.

usrdata

A pointer to the user data to associate with the dictionary.

Library:

mmrndclient

Description:

Set a pointer to the user data to associate with the dictionary returned with the last event. The dictionary is stored in the **mmr_event_t** structure's *data* field and contains all the **mm-renderer** properties reported by the event.

Some event types, including STATE, ERROR, and WARNING, share a single dictionary and therefore have a common user data pointer. So, if you set the user data after receiving, say, a STATE event, the same user data pointer is returned with any subsequent STATE, ERROR, or WARNING event. Other event types, including METADATA, OUTPUT, and TRKPAR, each have multiple dictionaries, distinguished by an index stored in the **mmr_event_t** *details* field. So, if you set the user data after receiving say, a METADATA event with an index of 2, the same user data is returned only for other METADATA events whose index is also 2.

Returns:

Zero on success, or -1 if the event was MMR_EVENT_NONE or a deletion.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No

Safety:	
Thread	**No**, except when using different context handles

Related Links

mmr_event_t (p. 70)
*Information about an **mm-renderer** event*

mmr_event_get()

Get the next available event

Synopsis:

```
#include <mm/renderer/events.h>

const mmr_event_t* mmr_event_get( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Get the next available event. The function returns an **mmr_event_t** structure, which contains detailed event information such as the new context state (see **mmr_event_t** (p. 70) for details). Typically, you would call this function within an event-processing loop, after calling either *mmr_event_arm()* (p. 64) or *mmr_event_wait()* (p. 80).

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

In any playback state, *mmr_event_get()* might return the MMR_EVENT_NONE event. Applications must gracefully handle this event, perhaps simply by ignoring it. For an example of a situation when MMR_EVENT_NONE might be returned, see *mmr_event_wait()* (p. 80).

Returns:

A pointer to an event, or NULL on failure (check *errno*).

Classification:

QNX Neutrino

Safety:

Interrupt handler	No
Signal handler	No
Thread	**No**, except when using different context handles

Related Links

mmr_event_t (p. 70)
*Information about an **mm-renderer** event*

mmr_event_t

*Information about an **mm-renderer** event*

Synopsis:

```
#include <mm/renderer/events.h>

typedef struct mmr_event {
    mmr_event_type_t type;
    mmr_state_t state;
    int speed;
    union mmr_event::mmr_event_details details;
    const strm_string_t *pos_obj;
    const char *pos_str;
    const strm_dict_t *data;
    const char *objname;
    void *usrdata;
} mmr_event_t;
```

Data:

type

The event type.

state

The new context state (valid even when type is MMR_EVENT_NONE).

speed

The playback speed (0 means paused).

details

The event details (varies by type).

pos_obj

The playback position when the event occurred, stored as a shareable string, for STATUS, ERROR, and WARNING events; otherwise NULL.

pos_str

The playback position when the event occurred, stored as a string, for STATUS, ERROR, and WARNING events; otherwise NULL.

The position is expressed in the same media-specific format used by *mmr_seek()* (p. 109) (for single tracks, it's `milliseconds`; for playlists, it's `tracknumber:milliseconds`; for autolists, it's `1:milliseconds`).

data

The full set of **mm-renderer** properties reported by the event, stored in a dictionary object. When this field is NULL, the set of properties no longer exists; for example, the input parameters, URL, and type are deleted when the input is detached.

objname

The name of the internal **mm-renderer** object associated with this event.

usrdata

The user data associated with the dictionary; NULL if there's no user data.

Library:

mmrndclient

Description:

The **mmr_event_t** structure is returned by *mmr_event_get()* (p. 68) and provides information on the last event. For all events, the structure contains the event type, the latest context state and playback speed, and if applicable, the playback position when the event occurred. For certain event types, there is additional information contained in the **mmr_event::details** and/or **mmr_event::data** fields.

Classification:

QNX Neutrino

Related Links

mmr_event_get() (p. 68)
Get the next available event

mmr_event::data

*Properties reported by an **mm-renderer** event*

Description:

All the **mm-renderer** properties reported by the event, represented as a dictionary object stored in the *data* field of the **mmr_event_t** data structure. Some events, such as STATE, ERROR, and WARNING events, have all their properties pre-parsed into other **mmr_event_t** fields, so no additional information can be found in the *data* field. Other events, notably events that indicate updates to **mm-renderer** parameters or metadata, have most of their properties stored in the *data* field, so clients must extract this information from the dictionary.

Which properties are stored in the dictionary depends also on the configuration of the attached input and outputs. For instance, the properties returned with events that indicate updates to input, track, or output parameters consist of the latest parameters the client passed into the Client API, plus the track URL and type. The properties returned with METADATA events vary with the input properties (e.g., the file format, and whether the input is a track or playlist).

To look up parameter values in a **strm_dict_t** dictionary object by name, see the *strm_dict_find_value()* (p. 119) function in the Dictionary Object API.

The event types with information stored in the *data* field, and the associated dictionary contents are as follows:

Event type	Property name	Description
MMR_EVENT_STATUS	*position*	Playback position. This is the same value stored in the *pos_str* (p. 70) field in the **mmr_event_t** structure.
	bufferstatus	Status or current activity of the buffer: `buffering`, `playing`, or `idle`
	bufferlevel	Buffer usage level, represented by two decimal numbers (in milliseconds) separated by a slash: `level/capacity`
	volume	Available only for audio recording. The current volume level of the input signal against the maximum volume level, separated by a slash: `current_level/maximum_level`
MMR_EVENT_METADATA	*md_title_name*	Track name
	md_title_artist	Artist name
	md_title_album	Album name
	md_title_genre	Genre (classical, rock, funk, etc.)
	md_title_comment	Text information about the track. The source of this information depends on the track format. For example, the information is taken from the COMM frame for ID3 tracks.
	md_title_duration	Track length

Event type	Property name	Description
	md_title_track	Track number (if track was ripped from CD)
	md_title_disc	Disc number (if track was ripped from multi-disc album)
	md_title_samplerate	Number of samples taken from input signal per time unit (typically, a second) during recording. For audio content, this field refers to the audio sampling rate; for video, it's the video frame rate.
	md_title_bitrate	Track bit rate. This is only an approximate value, based on the total bit rate from all media streams in the input track and ignoring any potential variation in bit rate throughout playback.
	md_title_protected	Boolean attribute (either 0 or 1) indicating if track is DRM-protected
	md_title_seekable	Boolean attribute (either 0 or 1) indicating if track supports seeking
	md_title_pausable	Boolean attribute (either 0 or 1) indicating if track can be paused
	md_title_mediatype	Track media type (2 for video only, 4 for audio only, and 6 for audio and video)
	md_video_width	Video width, in physical units
	md_video_height	Video height, in physical units
	md_video_pixel_width	Video width, in pixels
	md_video_pixel_height	Video height, in pixels
	md_video_capture_format	Media file format in which video was recorded
MMR_EVENT_PLAYLIST	All playlist information fields	Playlist information fields contained in the **mmr_event_playlist** (p. 75) structure, which is stored in the *details* field of the **mmr_event** structure for playlist events.
MMR_EVENT_INPUT	All input parameters and the input URL and type	Input parameters specified in the last call to *mmr_input_parameters()* (p. 89), and the input URL and type specified in the last call to *mmr_input_attach()* (p. 84). Some input parameters may have been changed by **mm-renderer**.
MMR_EVENT_OUTPUT	All output parameters and the output URL and type	Output parameters specified in the last call to *mmr_output_parameters()* (p. 101), the output type specified in the last call to *mmr_output_attach()* (p. 96), and the output URL specified in the last call to either of these two functions.
MMR_EVENT_CTXTPAR	All context parameters	Context parameters specified in the last call to *mmr_context_parameters()* (p. 52).
MMR_EVENT_TRKPAR	All track parameters	Track parameters specified in the last call to *mmr_track_parameters()* (p. 92).

Related Links

mmr_event_type_t (p. 78)
*The **mm-renderer** events reported by the API*

strm_dict_find_value() (p. 119)
Find the value of a dictionary entry based on the entry's key (returns a string)

mmr_event::details

*Details for an **mm-renderer** event*

Synopsis:

```
#include <mm/renderer/events.h>

union mmr_event::mmr_event_details {

    struct mmr_event_state {
        mmr_state_t oldstate;
        int oldspeed;
    } state;

    struct mmr_event_error {
        mmr_error_info_t info;
    } error;

    struct mmr_event_warning {
        const char *str;
        const strm_string_t *obj;
    } warning;

    struct mmr_event_metadata {
        unsigned index;
    } metadata;

    struct mmr_event_trkparam {
        unsigned index;
    } trkparam;

    struct mmr_event_playlist {
        unsigned start;
        unsigned end;
        unsigned length;
    } playlist;

    struct mmr_event_output {
        unsigned id;
    } output;

} details;
```

Data:

state

Used when **mmr_event.type** is MMR_EVENT_STATE.

The **mmr_event_state** structure has these members:

mmr_state_t ***oldstate***

The state before the event.

int ***oldspeed***

The speed before the event.

error

Used when **mmr_event.type** is MMR_EVENT_ERROR.

The **mmr_event_error** structure has these members:

mmr_error_info_t ***info***

The error information.

warning

Used when **mmr_event.type** is MMR_EVENT_WARNING.

The **mmr_event_warning** structure has these members:

const char* ***str***

The warning string, as a C string.

const strm_string_t* ***obj***

The warning string, as a strm_string_t (dictionary string).

metadata

Used when **mmr_event.type** is MMR_EVENT_METADATA.

The **mmr_event_metadata** structure has these members:

unsigned ***index***

The playlist index for playlist-related events; otherwise, zero.

trkparam

Used when **mmr_event.type** is MMR_EVENT_TRKPAR.

The **mmr_event_trkparam** structure has these members:

unsigned ***index***

The playlist index.

playlist

Used when **mmr_event.type** is MMR_EVENT_PLAYLIST.

The **mmr_event_playlist** structure has these members:

unsigned *start*

The index of the first item in the playlist window.

unsigned *end*

The index of the last item in the playlist window.

unsigned *length*

The playlist length.

output

Used when **mmr_event.type** is MMR_EVENT_OUTPUT.

The **mmr_event_output** structure has these members:

unsigned *id*

The output ID.

Library:

mmrndclient

Classification:

QNX Neutrino

Related Links

mmr_event_type_t (p. 78)

*The **mm-renderer** events reported by the API*

mmr_event_type_t

*The **mm-renderer** events reported by the API*

Synopsis:

```
#include <mm/renderer/events.h>

typedef enum mmr_event_type {
    MMR_EVENT_NONE,
    MMR_EVENT_ERROR,
    MMR_EVENT_STATE,
    MMR_EVENT_OVERFLOW,
    MMR_EVENT_WARNING,
    MMR_EVENT_STATUS,
    MMR_EVENT_METADATA,
    MMR_EVENT_PLAYLIST,
    MMR_EVENT_INPUT,
    MMR_EVENT_OUTPUT,
    MMR_EVENT_CTXTPAR,
    MMR_EVENT_TRKPAR,
    MMR_EVENT_OTHER,
} mmr_event_type_t;
```

Data:

MMR_EVENT_NONE

No pending events.

MMR_EVENT_ERROR

Playback has stopped due to an error or EOF.

MMR_EVENT_STATE

State or speed change, other than an error or EOF.

MMR_EVENT_OVERFLOW

Some state changes lost; the event contains the most recent state.

MMR_EVENT_WARNING

Warning event.

MMR_EVENT_STATUS

Status update (position, buffer level, etc).

MMR_EVENT_METADATA

Metadata update for the attached input, or one track referenced by the attached input (such as a playlist entry).

MMR_EVENT_PLAYLIST

Playlist window update.

MMR_EVENT_INPUT

An input has been attached or detached, or input parameters have changed.

MMR_EVENT_OUTPUT

An output has been attached or detached, or output parameters have changed.

MMR_EVENT_CTXTPAR

Context parameters have changed.

MMR_EVENT_TRKPAR

Track parameters for an individual track or a playlist entry have changed.

MMR_EVENT_OTHER

None of the above, but something has changed. You can typically ignore this event type.

Library:

mmrndclient

Description:

The enumerated type **mmr_event_type_t** defines all possible events that can be observed through the Event API. Events include: changes to the context state or playback speed; updates of metadata or the playlist window; and attachment and detachment of input and output devices.

To obtain the type of the last event, call *mmr_event_get()* and examine the **type** field in the **mmr_event_t** structure returned by the function.

Classification:

QNX Neutrino

Related Links

mmr_event_get() (p. 68)
Get the next available event

mmr_event_wait()

Wait until an event is available

Synopsis:

```
#include <mm/renderer/events.h>

int mmr_event_wait( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Wait for an event. This function usually blocks until an event occurs, at which point it unblocks and you can call *mmr_event_get()* (p. 68) to get the event details.

Occasionally, *mmr_event_wait()* may unblock or not block at all even though no events are available. For example, suppose a track enters the playlist range but then exits soon afterwards. The **mm-renderer** service creates metadata for the track when it comes in range and this activity generates an event. If the track exits the playlist range before the application calls *mmr_event_get()*, the track's metadata and the corresponding event are deleted. In the subsequent call to *mmr_event_get()*, the function will return the MMR_EVENT_NONE event.

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

Returns:

Zero on success, or -1 on failure (check *errno*).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	**No**, except when using different context handles

Related Links

mmr_event_t (p. 70)
*Information about an **mm-renderer** event*

mmr_state_t

The context states

Synopsis:

```
#include <mm/renderer/events.h>

typedef enum mmr_state {
    MMR_STATE_DESTROYED,
    MMR_STATE_IDLE,
    MMR_STATE_STOPPED,
    MMR_STATE_PLAYING,
} mmr_state_t;
```

Data:

MMR_STATE_DESTROYED

The context has been destroyed.

MMR_STATE_IDLE

The context has no input.

MMR_STATE_STOPPED

The context has an input but is not playing.

MMR_STATE_PLAYING

The context is playing or paused.

Library:

mmrndclient

Description:

The enumerated type **mmr_state_t** defines the context states, which are based on the current input and playback activity.

To obtain the context state following the latest API event, call *mmr_event_get()* and examine the **state** field in the **mmr_event_t** structure returned by the function.

Classification:

QNX Neutrino

Related Links

mmr_event_get() (p. 68)
Get the next available event

Input configuration

You can attach an input to the context by naming the URL of a track or playlist on a media device or in the local filesystem. The acceptable URL formats vary with the input type. For some formats, you can include parameters in the URL string, to specify settings such as the sampling rate for audio capture devices.

A context can have only one input; if you call *mmr_input_attach()* once to attach an input and then call it later to attach a new input without having called *mmr_input_detach()*, the first input gets detached when the second one is attached.

With *mmr_input_parameters()*, you can instruct **mm-renderer** to repeat playback of a single track or many tracks in sequence. For inputs that use the Audio Manager service, you can specify audio stream characteristics. For HTTP-based inputs, you can configure settings for cookies, proxy servers, and authentication credentials.

Finally, for playlists, you can override any of the input parameters (except the repeat setting) for individual tracks within the playlist by calling *mmr_track_parameters()*.

mmr_input_attach()

Attach an input

Synopsis:

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

int mmr_input_attach( mmr_context_t *ctxt,
                      const char *url,
                      const char *type )
```

Arguments:

ctxt

A context handle.

url

The URL of the new input.

type

The input type. You must place quotes around the text naming the input type, which can be one of the following:

`track`

One track played in isolation from the rest of the media

`playlist`

A track sequence, with ordering information and track metadata contained in a playlist file

`autolist`

A single track formatted as a playlist; you can play the track continuously using the *repeat* parameter

Library:

mmrndclient

Description:

Attach an input track or playlist. If the context already has an input, the function detaches it first.

The input type affects how **mm-renderer** responds to certain playback requests. For example, when seeking to track positions using *mmr_seek()* (p. 109), you must specify the desired position differently for each of the supported input types. Also, *mmr_list_change()* (p. 106) and *mmr_track_parameters()* (p. 92) apply to "`playlist`" only.

Which input types are supported depends on the configuration of **mm-renderer**; however, the playback behavior for a given input type does *not* depend on the configuration.

Valid input URLs for the `"track"` and `"autolist"` input types are:

- An `http:` or `https:` URL containing the full path of an HTTP-accessible file or an HTTP stream. When the URL refers to a file, the `http:` prefix is optional.

  HTTP Live Streaming (HLS) is supported just as any HTTP stream, with the following caveats:

  - For HLS realtime broadcast, the seek operation is disabled. Therefore, if your application issues a seek command it will fail.
  - Pause (play speed of 0) is supported but the playback may jump forward when resumed because the current stream may have become unavailable.
  - For HLS Video on Demand, the seek operation places the play position at the start of the video chunk that is closest to the requested time. The pause operation works as expected.

  The **mm-renderer** service supports HLS version 3, with media segments encoded as follows:

  - Transport stream MPEG2-TS with H.264 Video, with either MP3 or AAC Audio (when the appropriate codecs are available on the platform)
  - Video only and audio only when embedded in the MPEG2-TS stream

  

  For secure video and audio playback of HTTP streams, **mm-renderer** allows you to set cookies, SSL, and authentication properties by defining *context parameters* (p. 52), *input parameters* (p. 89), or *track parameters* (p. 92).

- An `rtsp:` URL naming an RTSP source, which will deliver video streamed over RTP. The source is indicated by either a host name or an IP address, followed by the path. To authenticate with the RTSP server, you can provide a username and password, separated by a colon and followed by the AT sign, in front of the source:

  ```
  rtsp://username:password@10.222.97.225/axis-media/media.amp
  ```

- An `rtp:` URL specifying a port on which to listen for a unicast RTP stream. The port number is prefixed with the AT sign and a colon, as in this example:

  ```
  rtp://@:49152
  ```

  This will listen for an RTP stream on port 49152. This URL type is useful if you want to configure a camera to stream to a QNX Neutrino host.
- A full pathname starting with a "/" character, with or without a `file:` prefix. The pathname may refer to an audio or a video file.
- A `file2b:` URL containing the full pathname of a dynamically growing file (i.e., a *progressive download*). Not all formats are supported. If parsing the file requires knowing its size or reading more data than currently in the file, the input attachment operation may fail. If it succeeds, any attempt to play from beyond the end of file will cause the playback to underrun. Your application must pay attention to the buffering status and appropriately present the state to the user, depending on whether the download is happening at the same time.

- An `snd:` URL targeting an audio capture device (microphone) whose device file is located in **/dev/snd**. The URL can specify device configuration options in a comma-separated list, as follows:

  ```
  snd:/dev/snd/pcmPreferredc?frate=44100&nchan=2
  ```

  Supported options include:

  - `frate` — the sampling rate, in Hertz
  - `frag_ms` — the fragment size, in milliseconds
  - `nchan` — the number of channels (1 for mono, 2 for stereo)
  - `depth` — the number of bits per sample (e.g., 16)
  - `bsize` — the preferred read size, in bytes

  Currently, this URL format works only with the "`file`" output type. The resulting behavior is identical to that for the `audio:` URL format except that the Audio Manager is bypassed, which means you can't provide hints such as audio type to control audio routing. The advantage of the `snd:` prefix is that you can use it on systems that don't have Audio Manager.
- An `audio:` URL naming an audio capture device (microphone) whose name is defined by the `AUDIO_DEVICE_NAMES` set of string constants, which is listed in the *Audio Manager Library* reference. The URL can specify any of the options supported for `snd:` URLs, for example:

  ```
  audio:voice?nchan=1&frate=44100&depth=16
  ```

  Currently, this URL format works only with the "`file`" output type. Client applications should use `audio:default` to specify automated routing for the audio stream unless there's a good reason to use one particular device. When a non-default device is named, the audio stream routing depends solely on that device. In this case, the removal of the device could result in no audio being outputted or an error returned to the client.

Not all audio devices may work with the current application. It's the client's responsibility to determine if a particular device is supported before trying to use it. See the *mmr_output_attach() example* (p. 98) for a demo of how to check if an audio device is supported before configuring the audio routing.

Valid input URLs for the "`playlist`" input type are:

- A full pathname of an M3U playlist file, with or without a `file:` or `http:` prefix
- An SQL URL in the form `sql:` *database* `?query=` *query*, where *database* is the full path to the database file, and *query* contains an SQL query that results in a single column containing URLs in a form acceptable for the "`track`" input type. Note that any special characters in the query must be URL-encoded (e.g., spaces encoded as `%20`).

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Example:

See the *mmr_output_attach() example* (p. 98) for a demo of how to select one audio device to use.

Related Links

mmr_output_attach() (p. 96)
Attach an output

AUDIO_DEVICE_NAMES

mmr_input_detach()

Detach an input

Synopsis:

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

int mmr_input_detach( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Detach any input. If the context is playing, stop it first. If there is no input already, this is a no-op.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_input_parameters()

Set input parameters

Synopsis:

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

int mmr_input_parameters( mmr_context_t *ctxt, strm_dict_t *parms )
```

Arguments:

ctxt

A context handle.

parms

A reference to a dictionary containing the input parameters to set (must not be NULL). Any previous parameters are overridden.

The **strm_dict_t** object becomes API property after this call, even if the call fails. You should not use or destroy the dictionary after passing it to this function.

Library:

mmrndclient

Description:

Set parameters associated with the attached input.

Some **mm-renderer** plugins don't return errors when you provide unacceptable values for input parameters. Instead, these plugins revert bad parameters to their previous values or to their default values (for parameters that you set for the first time). To see which values were accepted or changed, client applications can examine the parameters that the Event API returned.

The input type and the URL format determine which input parameters you can set. Some parameters must be set before the input is attached, because setting them after attaching the input has no effect. All input parameters are cleared when the input is detached, whether explicitly through a call to *mmr_input_detach()* or implicitly when *mmr_input_attach()* is called to attach a new input, which causes **mm-renderer** to detach the current input.

An individual playlist item isn't considered an input but the whole playlist is. For playlists, any input parameters that you set will apply to the playlist file. To set parameters for individual playlist items, you must use *mmr_track_parameters()*.

The `"playlist"` and `"autolist"` input types support the following parameter:

repeat

Continuously replay the input. Acceptable values are `"none"` (default), `"track"`, or `"all"`.

When using the "`autolist`" or "`track`" input types with a URL that starts with `audio:`, you can set one of the following two parameters:

audio_type

Classify the audio input based on its content (voice, ring tones, video chat, etc.). This parameter provides a shortcut for setting the audio type, thereby simplifying your client code. You can use this parameter instead of using the Audio Manager API to obtain an audio manager handle, and then using that handle to set the audio type.

The audio type is specified as a string that's set to one of the audio types defined by `AUDIO_TYPE_NAMES`, which is documented in the *Audio Manager Library* reference.

audioman_handle

Associate an audio manager handle with the audio stream that the current context manages. To obtain a value for this parameter, call the *audio_manager_get_handle()* API function and pass in the desired audio type.

You can then use this handle to change the audio type and other audio stream characteristics through the Audio Manager API. For more information, refer to the audio routing functions described in the *Audio Manager Library* reference.

When the input URL starts with `http:` or `https:`, you can set the following parameters that map to **libcurl** options:

- *OPT_VERBOSE*
- *OPT_CONNECTTIMEOUT_MS*
- *OPT_LOW_SPEED_LIMIT*
- *OPT_LOW_SPEED_TIME*
- *OPT_USERAGENT*
- *OPT_USERNAME*
- *OPT_PASSWORD*
- *OPT_PROXYUSERNAME*
- *OPT_PROXYPASSWORD*
- *OPT_COOKIE*
- *OPT_COOKIEFILE*
- *OPT_COOKIEJAR*
- *OPT_COOKIESESSION*
- *OPT_CAINFO*
- *OPT_CAPATH*
- *OPT_SSL_VERIFYPEER*
- *OPT_SSL_VERIFYHOST*
- *OPT_PROXY*
- *OPT_NOPROXY*
- *OPT_HTTPPROXYTUNNEL*
- *OPT_PROXYPORT*
- *OPT_PROXYTYPE*
- *OPT_PROXYAUTH*

- *OPT_HTTPAUTH*
- *OPT_HTTPHEADER*
- *OPT_DNSCACHETIMEOUT*

You can set these same **libcurl** options through the context parameters. For any options defined in both parameter sets, the input parameter settings take precedence.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Examples:

See the *mmr_output_parameters() examples* (p. 102) for demos on how to set each of the *audio_type* and *audioman_handle* parameters.

Related Links

Defining Parameters (p. 26)
Parameters allow you to set various properties that influence how media files are accessed and rendered during playback.

AUDIO_TYPE_NAMES

audio_manager_get_handle()

audio routing functions

strm_dict_t (p. 131)
Dictionary object type

mmr_track_parameters()

Set track parameters

Synopsis:

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

int mmr_track_parameters( mmr_context_t *ctxt,
                          unsigned index,
                          strm_dict_t *parms )
```

Arguments:

ctxt

A context handle.

index

Zero to set the default parameters, or a nonzero index within the current playlist window.

parms

A reference to a dictionary containing the track parameters to set. Use NULL to reset the parameters of the specified track to the default values assigned to track `0`. Any previous parameters are overridden.

The **strm_dict_t** object becomes API property after this call, even if the call fails. You should not use or destroy the dictionary after passing it to this function.

Library:

mmrndclient

Description:

Set track parameters. This function can be used when the input type is "`playlist`" or "`autolist`". When the input type is "`track`", this function has no effect.

For "`playlist`" inputs, *index* specifies the track that these parameters are applied to. The provided index must be within range of the current playlist window or the function call will fail. An index of zero specifies the default parameters given to a new track when it enters the playlist window.

For "`autolist`" inputs, any input parameters that you set before attaching the input are taken as the initial track parameters (because the single track is the input). If you want to change them after attaching the input, use *mmr_track_parameters()*. Changes to input parameters other than *repeat* are ignored.

Some **mm-renderer** plugins don't return errors when you provide unacceptable values for track parameters. Instead, these plugins revert bad parameters to their previous values or to their default values (for parameters that you set for the first time). To see which values were accepted or changed, client applications can examine the parameters that the Event API returned.

When the input URL starts with `audio:`, you can set one of the following two parameters:

audio_type

Classify the audio track based on its content (voice, ring tones, video chat, etc.). This parameter provides a shortcut for setting the audio type, thereby simplifying your client code. You can use this parameter instead of using the Audio Manager API to obtain an audio manager handle, and then using that handle to set the audio type.

The audio type is specified as a string that's set to one of the audio types defined by `AUDIO_TYPE_NAMES`, which is documented in the *Audio Manager Library* reference.

audioman_handle

Associate an audio manager handle with the audio stream that the current context manages. To obtain a value for this parameter, call the *audio_manager_get_handle()* API function and pass in the desired audio type.

You can then use this handle to change the audio type and other audio stream characteristics through the Audio Manager API. For more information, refer to the audio routing functions described in the *Audio Manager Library* reference.

When the input URL starts with `http:` or `https:`, you can set the following parameters that map to **libcurl** options:

- *OPT_VERBOSE*
- *OPT_CONNECTTIMEOUT_MS*
- *OPT_LOW_SPEED_LIMIT*
- *OPT_LOW_SPEED_TIME*
- *OPT_USERAGENT*
- *OPT_USERNAME*
- *OPT_PASSWORD*
- *OPT_PROXYUSERNAME*
- *OPT_PROXYPASSWORD*
- *OPT_COOKIE*
- *OPT_COOKIEFILE*
- *OPT_COOKIEJAR*
- *OPT_COOKIESESSION*
- *OPT_CAINFO*
- *OPT_CAPATH*
- *OPT_SSL_VERIFYPEER*
- *OPT_SSL_VERIFYHOST*
- *OPT_PROXY*
- *OPT_NOPROXY*
- *OPT_HTTPPROXYTUNNEL*
- *OPT_PROXYPORT*
- *OPT_PROXYTYPE*
- *OPT_PROXYAUTH*
- *OPT_HTTPAUTH*

- *OPT_HTTPHEADER*
- *OPT_DNSCACHETIMEOUT*

You can set these same **libcurl** options through the context or input parameters. For any options defined in either the context or input parameters but also in the track parameters, the track parameter settings take precedence.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Examples:

See the *mmr_output_parameters() examples* (p. 102) for demos on how to set each of the *audio_type* and *audioman_handle* parameters.

Related Links

Defining Parameters (p. 26)
Parameters allow you to set various properties that influence how media files are accessed and rendered during playback.

AUDIO_TYPE_NAMES

audio_manager_get_handle()

audio routing functions

strm_dict_t (p. 131)
Dictionary object type

Output configuration

You must attach an output to a context before attaching the input. An output is specified by naming a URL. The acceptable URL format depends on the output type, which can be an audio device, video device, or file. For video outputs, the URL string can configure settings of the output window.

A context can have multiple outputs; for instance, when playing video files, you can attach both a video device and an audio device as outputs by making separate calls to *mmr_output_attach()*. Individual outputs can be detached with *mmr_output_detach()*.

With *mmr_output_parameters()*, you can set the volume for any audio output. For those outputs that use the Audio Manager service, you can also specify audio stream characteristics. Note that the Screen API, not the **mm-renderer** API, should be used for manipulating properties of video output. There are no output parameters for file outputs.

mmr_output_attach()

Attach an output

Synopsis:

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

int mmr_output_attach( mmr_context_t *ctxt,
                       const char *url,
                       const char *type )
```

Arguments:

ctxt

A context handle.

url

The URL of the new output.

type

The output type. Possible values are "audio", "video", and "file" (quotes are required).

Library:

mmrndclient

Description:

Attach an output to the context and return its output ID (a non-negative integer, unique for this context). An output can be an audio or video device, a combined audio/video device (such as a DSP directly connected to output hardware), or a file. The types of outputs attached to a context may affect the set of operations that the context will allow. For instance, when "playing" to a file (i.e., ripping), seeking or trick play (i.e., playing at non-normal speeds) may not be supported.

Although the API allows requesting multiple outputs of the same type, this may not be supported by all player module implementations. Attaching or detaching outputs while the context has an input may not be supported, either.

Valid URLs for the "audio" output type are in one of the following forms:

- snd:*device*, where *device* is the path of an audio output device, such as **/dev/snd/pcmPreferredp**. This URL format produces behavior similar to the audio: format, except that the Audio Manager is bypassed and so you can't provide hints such as the audio type to control audio routing. The advantage of the snd: prefix is that you can use it on systems that don't have Audio Manager.
- audio:*name*, where *name* is one of the audio device names defined by the AUDIO_DEVICE_NAMES set of string constants, which is listed in the *Audio Manager Library* reference.

The **mm-renderer** service opens the device named in the URL. Client applications should use `audio:default` to specify automated routing for the audio stream unless there's a good reason to use one particular device. When a non-default device is named, the audio stream routing depends solely on that device. In this case, the removal of the device could result in no audio being outputted, or an error returned to the client.

Not all defined audio devices may work with the current application. It is the client's responsibility to determine if a particular device is supported before trying to use it. See the *example* (p. 98) section for a demo of how to check if an audio device is supported before configuring the audio routing.

Valid output URLs for the "`video`" output type are of the following form:

```
screen:?wingrp=window_group&
  winid=window_id&nodstviewport=1&initflags=invisible
```

In the video URL:

- *window_group* is the window group name of the application's top-level window
- *window_id* is the window ID for the window where the video output will be rendered
- The parameter setting `nodstviewport=1` is optional, and forces **mm-renderer** to never directly modify the destination viewport of the window. This avoids conflicts between simultaneous application manipulation and **mm-renderer** manipulation of the destination viewport.
- The parameter setting `initflags=invisible` is optional, and causes the window to be invisible upon creation. This flag allows you to adjust window properties such as size, position, and z-order before making it visible.

Valid output URLs for the "`file`" output type are of the form `file:` *path*, where *path* is the full filepath. The `file:` prefix is optional. The following file types (and their extensions that can go in the URL) are supported:

- Waveform Audio File Format (**.wav**)
- MPEG 4 Audio (**.m4a**)
- Adaptive Multi-Rate (**.amr**)
- 3GPP file format (**.3gp**)
- Adaptive Multi-Rate Wideband (**.awb**)
- Qualcomm PureVoice (**.qcp**)

Returns:

A non-negative output ID on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:

Interrupt handler	No

Safety:	
Signal handler	No
Thread	Yes

Example:

Suppose you want **mm-renderer** to route the audio output to the "speaker" device. You can do this as follows:

```
#include <mm/renderer.h>
#include <audio/audio_manager_device.h>

// Point the output URL to the default device
char audio_URL[100];
snprintf( audio_URL, 100, "audio:default" );

// Check if the speaker device is supported and
// connected to the system; if so, point the
// output URL to the speaker device
bool supported, connected;

if ( audio_manager_is_device_supported(
                        AUDIO_DEVICE_SPEAKER,
                        &supported ) == EOK
                                  && supported )
{
    if ( audio_manager_is_device_connected(
                        AUDIO_DEVICE_SPEAKER,
                        &connected ) == EOK
                                  && connected )
    {
        sprintf( audio_URL, "audio:%s",
            audio_manager_get_device_name(
                        AUDIO_DEVICE_SPEAKER ) );
    }
}

// Attach an audio output, routed to the
// chosen device
int outputID;

if ( outputID = mmr_output_attach( context,
                        audio_URL, "audio" ) < 0 )
{
    // Call mmr_error_info() and do error handling
}
```

You can make **mm-renderer** read the input audio from a particular device in a similar way, by substituting the call to *mmr_output_attach()* with a call to *mmr_input_attach()*. The input URL would be of the same form.

Related Links

AUDIO_DEVICE_NAMES

mmr_output_detach()

Detach an output

Synopsis:

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

int mmr_output_detach( mmr_context_t *ctxt, unsigned output_id )
```

Arguments:

ctxt

A context handle.

output_id

An output ID.

Library:

mmrndclient

Description:

Detach the specified output.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_output_parameters()

Set output parameters

Synopsis:

```
#include <mm/renderer.h>
#include <audio/audio_manager_routing.h>

int mmr_output_parameters( mmr_context_t *ctxt,
                           unsigned output_id,
                           strm_dict_t *parms )
```

Arguments:

ctxt

A context handle.

output_id

An output ID.

parms

A reference to a dictionary containing the output parameters to set (must not be NULL). Any previous parameters are overridden.

The **strm_dict_t** object becomes API property after this call, even if the call fails. You should not use or destroy the dictionary after passing it to this function.

Library:

mmrndclient, audio_manager_lib

Description:

Set parameters for an output device. The acceptable parameter values depend on the plugins loaded for the attached output and the attached input, if any. Unlike input and track parameters, the values of output parameters won't be changed by **mm-renderer** plugins. If the provided values aren't supported for the current output and input combination, the function call fails.

The output type determines which output parameters you can set. At present, there are no output parameters for the "`file`" output type.

For the "`audio`" output type, the following parameter is available for any URL format:

volume

Set the volume for this audio stream. The volume must be in the range of 0 to 100.

When using the "`audio`" output type with a URL that starts with `audio:`, you can set one of the following two parameters:

audio_type

Classify the audio track based on its content (voice, ring tones, video chat, etc.). This parameter provides a shortcut for setting the audio type, thereby simplifying your client code. You can use this parameter instead of using the Audio Manager API to obtain an audio manager handle, and then using that handle to set the audio type.

The audio type is specified as a string that's set to one of the audio types defined by `AUDIO_TYPE_NAMES`, which is documented in the *Audio Manager Library* reference.

audioman_handle

Associate an audio manager handle with the audio stream that the current context manages. To obtain a value for this parameter, call the *audio_manager_get_handle()* API function and pass in the desired audio type.

You can then use this handle to change the audio type and other audio stream characteristics through the Audio Manager API. For more information, refer to the audio routing functions described in the *Audio Manager Library* reference.

For the `"video"` output type, your application should modify the output window directly by using the **libscreen** library, as demonstrated in "*Managing video windows* (p. 28)".

CAUTION: The legacy video output parameters *video_dest_**, *video_src_**, and *video_clip_** have been deprecated. Using **libscreen** is the proper way to configure video output.

The *mmr_output_attach()* function sets the parameters *url* and *type*. Some plugins allow you to modify the URL with *mmr_output_parameters()*. For instance, you can ask **mm-renderer** to switch output devices by calling *mmr_output_parameters()* with a new URL in the parameters.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:

Interrupt handler	No
Signal handler	No
Thread	Yes

Examples:

Suppose you want to set the *audio_type* parameter to indicate that an output audio stream contains dialing and call progress tones, also referred to as *voice tones*. You must look up the audio type string

by passing the AUDIO_TYPES_VOICE_TONES code to the Audio Manager API, store the returned string in a dictionary, and pass the dictionary to **mm-renderer**, as follows:

```
#include <mm/renderer.h>
#include <strm.h>

strm_dict_t* dict = strm_dict_new();

if ( dict = strm_dict_set( dict, "audio_type",
              audio_manager_get_name_from_type(
              AUDIO_TYPE_VOICE_TONES ) ) == NULL )
{
    // Do error handling
}

if ( mmr_output_parameters( context,
                            output_id, dict ) < 0 )
{
    // Call mmr_error_info() and do error handling
}
```

Presently, the audio type is the only audio stream characteristic that clients can set directly through **mm-renderer**. The Audio Manager API lets clients manage additional characteristics of an audio stream. For example, you could set both the audio type and reset conditions, as follows:

```
#include <mm/renderer.h>
#include <strm.h>
#include <audio/audio_manager_routing.h>

unsigned int audio_hndl;

if ( audio_manager_get_handle(
              AUDIO_TYPE_VOICE_TONES, 0,
              false, &audio_hndl ) != EOK )
{
    // Check errno, do error handling, and exit
}

if ( audio_manager_set_handle_routing_conditions(
       audio_hndl,
     SETTINGS_RESET_ON_DEVICE_CONNECTION ) != EOK )
{
    // Check errno, do error handling, and exit
}

// Store the handle in the dictionary before
```

```
// setting the output parameters
...
```

You can set the *audio_type* or *audioman_handle* parameters for an input in a similar way, by substituting the call to *mmr_output_parameters()* with a call to *mmr_input_parameters()*.

Related Links

Defining Parameters (p. 26)
Parameters allow you to set various properties that influence how media files are accessed and rendered during playback.

Managing video windows (p. 28)

AUDIO_TYPE_NAMES

audio_manager_get_handle()

audio routing functions

strm_dict_t (p. 131)
Dictionary object type

Playback control

After you've attached and configured the input and outputs for a context, you can send it playback commands to start and stop playback, adjust playback speed, and change the track position.

The *mmr_play()* and *mmr_stop()* functions start and stop the flow of media content from the input to the outputs. You can adjust the playback speed, including setting a speed of 0 to pause playback, with *mmr_speed_set()*.

You change the playback position with *mmr_seek()*, for all input types; however, the format of the string containing the position to seek to varies with the input type. For playlists, you can call *mmr_list_change()* during playback to switch to another playlist while continuing to play the current track.

mmr_list_change()

Set a new playlist

Synopsis:

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

int mmr_list_change( mmr_context_t *ctxt, const char *url, int delta )
```

Arguments:

ctxt

A context handle.

url

The URL of a new playlist.

delta

The difference between the position of the current track on the two lists.

Library:

mmrndclient

Description:

Set a new playlist without interrupting playback. This function can be used only during playback of a playlist (including when it's paused, but not stopped). The new playlist must contain the currently playing track at position *n+delta*, where *n* is its position on the old playlist. Note that *delta* is a signed value, so it can be negative.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60).

Common errors returned by this function and recommended follow-up actions are:

MMR_ERROR_INVALID_STATE

The context was stopped, not playing. The playback might have reached the end of the old playlist and so it was too late to switch playlists without interrupting playback. To fix, attach the new playlist as an input, seek to the beginning of the appropriate track, and start playback.

MMR_ERROR_INVALID_PARAMETER

The location in the new list you indicated (*n + delta*) is out of range or refers to a different URL. This error could be caused by a stale *delta* value, which results when the track you thought was playing just ended and a different track is playing now. To fix, recompute the difference between the position of the current track on the two lists.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_play()

Start playing

Synopsis:

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

int mmr_play( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Start playing. A no-op if already playing.

Once **mm-renderer** is playing media, you can adjust the play speed, seek to another position, change playlists, or stop playback.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

mmr_seek()

Seek to a position

Synopsis:

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

int mmr_seek( mmr_context_t *ctxt, const char *position )
```

Arguments:

ctxt

A context handle.

position

The position to seek to, in a media-specific format.

Library:

mmrndclient

Description:

Seek to a known position in a single track or a track within a playlist. The required format of the position string depends on the type of the attached input.

For the "track" type, the position is simply the number of milliseconds from the start of the track (e.g., "2500"). We refer to this time measurement as the *track offset*.

For the "playlist" and "autolist" types, the position must be specified as two numbers separated by a colon (e.g., "2:1200"), where the first number is the track index within the playlist and the second number is the track offset. For an "autolist" input, the first number must be 1.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Seeking to positions (p. 28)

mmr_speed_set()

Set the play speed, in units of 1/1000 of normal speed

Synopsis:

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

int mmr_speed_set( mmr_context_t *ctxt, int speed )
```

Arguments:

ctxt

A context handle.

speed

The new speed.

Library:

mmrndclient

Description:

Set the play speed, in units of 1/1000 of normal speed. If the context is playing (including if it's paused), the new speed is applied immediately; otherwise, it's stored in the context and applied the next time *mmr_play()* (p. 108) is called.

Use a speed of zero (0) to pause playback. Depending on the input media, *trick play*, which entails playing at speeds other than normal speed (1000), may be unsupported or forbidden, either completely or only for some portions of the media. Examples of this include:

- A playlist may contain tracks that don't support trick play
- Some devices have only one fast-forward speed
- DVDs forbid pausing or fast-forwarding through menus and some portions of titles

If an *mmr_speed_set()* call requests a speed for trick play but the exact value is completely unsupported by the current input, the speed may be rounded to a supported value in the same category (e.g., negative, slow, or fast). If the entire category is unsupported, the call fails.

If the call is made during playback and the speed (after the rounding described above) is unsupported or forbidden at the current position, the speed is changed to an allowed value and the call succeeds. A similar speed change may occur in the *mmr_play()* call, based on the curent speed and position, or during playback if a position is reached (by playing or by an explicit seek request) where the current play speed is unsupported or forbidden.

The navigation rules for the input media may also specify other circumstances that cause the speed to change to normal during playback. In particular, you can configure whether the speed reverts to normal when track boundaries are reached.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Play speed (p. 27)

Playing media (p. 27)
Playing media in **mm-renderer** requires configuring a context, attaching outputs and an input, and then issuing playback commands.

mmr_stop()

Stop playing

Synopsis:

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

int mmr_stop( mmr_context_t *ctxt )
```

Arguments:

ctxt

A context handle.

Library:

mmrndclient

Description:

Stop playing. A no-op if already stopped. Depending on the input media, stopping the playback may cause the playing position to change or even become indeterminate. Unless you know the behavior of the media being played, use the function *mmr_seek()* (p. 109) to seek to a known position before restarting the playback.

When playback is explicitly stopped using *mmr_stop()*, **mm-renderer** doesn't publish an error code. When the end of media is reached, the error code is set to MMR_ERROR_NONE.

Returns:

Zero on success, -1 on failure (use *mmr_error_info()* (p. 60)).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Chapter 4
Dictionary Object API

A dictionary object is a collection of key-value pairs that maps the names of parameters to their values. For **mm-renderer**, you can use the dictionary API to define context, input, and output parameters. Other components can use the same API to manage parameters specific to their purpose.

Both the keys and values are represented by shareable string objects. A shareable string is a data structure that encapsulates a string so that it can't be modified directly. This design allows multiple processes to read the string without the risk that the string will be changed inadvertently between reads. Modifying a shareable string actually destroys it and creates a new one.

The dictionary object API allows you to create multiple handles to a dictionary object and then use and even delete these handles in independent program components. After it's created, a dictionary is immutable until destroyed, so separate components can access it through their handles without worrying about synchronization.

Different dictionary object handles may be represented by identical pointers, so you shouldn't compare handles. Regardless of how the handles are represented internally, you must destroy each handle separately to properly dispose of any resources associated with it.

Related Links

strm_dict_clone() (p. 114)
Duplicate a dictionary handle

strm_dict_new() (p. 124)
Create a new handle for an empty dictionary object

strm_dict_set() (p. 125)
Modify a dictionary entry (using key-value strings)

strm_dict_clone()

Duplicate a dictionary handle

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_clone( const strm_dict_t *dict );
```

Arguments:

dict

A dictionary object handle.

Library:

libstrm

Description:

The function *strm_dict_clone()* creates a new handle to the dictionary object referenced by the specified handle.

Note that multiple handles may be represented by identical pointer values; you should not compare handles.

Returns:

A new dictionary object handle to the dictionary object referenced by the specified handle.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Dictionary Object API (p. 113)
A dictionary object is a collection of key-value pairs that maps the names of parameters to their values. For **mm-renderer**, you can use the dictionary API to define context, input, and output parameters. Other components can use the same API to manage parameters specific to their purpose.

strm_dict_compare()

Compare two dictionaries

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_compare( strm_dict_t *newdict,
                               strm_dict_t const *olddict );
```

Arguments:

newdict

A handle to the newer version of a dictionary object.

olddict

A handle to the older version of a dictionary object.

Library:

libstrm

Description:

The function *strm_dict_compare()* compares two dictionaries. It creates a replica of *newdict* and removes all those entries that also exist in *olddict* and have the same value. In other words, if *olddict* is the older version of some dictionary and *newdict* is the newer version, the resulting dictionary contains the entries that were changed or added, but not the ones that were left alone or deleted. If the same handle is passed for both arguments, this function returns an empty dictionary object.

Note that the *newdict* handle is closed and becomes invalid after calling this function, even on a failure, but the *olddict* handle is not (unless it's the same handle).

Returns:

A handle to the dictionary object containing the result of the comparison, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_destroy()

Destroy a dictionary object handle

Synopsis:

```
#include <strm.h>

int strm_dict_destroy( strm_dict_t *dict );
```

Arguments:

dict

A dictionary object handle.

Library:

libstrm

Description:

The function *strm_dict_destroy()* destroys the specified dictionary object handle and frees the memory allocated for the dictionary object if this is the last handle.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_find_index()

Return the index of a dictionary entry

Synopsis:

```
#include <strm.h>

ssize_t strm_dict_find_index( const strm_dict_t *dict, const char *key );
```

Arguments:

dict

A dictionary object handle.

key

The key of an entry to look up.

Library:

libstrm

Description:

The function *strm_dict_find_index()* returns the index of the entry specified by the *key* argument, if found in the specified dictionary.

Returns:

The index of the specified entry on success, or `-1` if the entry is not found.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_find_rstr()

Find the value of a dictionary entry based on the entry's key (returns a shareable string object handle)

Synopsis:

```
#include <strm.h>

const strm_string_t *strm_dict_find_rstr( const strm_dict_t *dict,
                                          const char *key );
```

Arguments:

dict

A handle to a dictionary object.

key

The name of the dictionary entry.

Library:

libstrm

Description:

The function *strm_dict_find_rstr()* finds the dictionary entry specified by the *key* argument, returning a handle to the entry's value. The returned shareable string object handle is owned by the dictionary, and remains valid until the dictionary handle is destroyed.

Returns:

A handle to the value of the dictionary entry specified by the *key* parameter if the the entry is found, or a null pointer if it isn't found.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_find_value()

Find the value of a dictionary entry based on the entry's key (returns a string)

Synopsis:

```
#include <strm.h>

const char *strm_dict_find_value( const strm_dict_t *dict, const char *key );
```

Arguments:

dict

A dictionary object handle.

key

The key of the dictionary entry.

Library:

libstrm

Description:

The function *strm_dict_find_value()* returns the value of the dictionary entry specified by the *key* argument. The returned string is owned by the dictionary, and remains valid until the dictionary handle is destroyed.

Returns:

The value (as a string) of the dictionary entry specified by the *key* parameter if the entry is found, or a null pointer if it isn't found.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_index_delete()

Delete a dictionary entry (by index)

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_index_delete( strm_dict_t *dict, size_t index );
```

Arguments:

dict

A dictionary object handle.

index

The index of the entry to delete.

Library:

libstrm

Description:

The function *strm_dict_index_delete()* creates a new dictionary object that is an exact replica of the old object, except the entry specified by the *index* argument is deleted. The function returns a handle for the new dictionary object, or a null pointer on failure (including when the index is out of range). On success, the original dictionary handle is destroyed.

Returns:

A handle for the new dictionary object, or a null pointer on failure (including when the index is out of range).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_key_delete()

Delete a dictionary entry (by key)

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_key_delete( strm_dict_t *dict, char const *key );
```

Arguments:

dict

A dictionary object handle.

key

The key of the entry to delete.

Library:

libstrm

Description:

The function *strm_dict_key_delete()* creates a new dictionary object that is an exact replica of the old object, except the entry specified by the *key* argument is deleted. The function returns a handle for the new dictionary object, or a null pointer on failure. On success, the original dictionary handle is destroyed.

Returns:

A handle for the new dictionary object, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_key_get()

Find the key of a dictionary entry (returns a string)

Synopsis:

```
#include <strm.h>

const char *strm_dict_key_get( const strm_dict_t *dict, size_t n );
```

Arguments:

dict

A dictionary object handle.

n

The 0-based index of the entry whose key is returned.

Library:

libstrm

Description:

The function *strm_dict_key_get()* finds the key of the (*n*+1)th entry in the specified dictionary and returns it as a null-terminated string. For example, if *n* is `3`, the key of the fourth entry is returned. The returned string is owned by the dictionary object, and remains valid until the dictionary handle is destroyed.

Returns:

The specified key on success, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:

Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_key_rstr()

Find the key of a dictionary entry (returns a shareable string object handle)

Synopsis:

```
#include <strm.h>

const strm_string_t *strm_dict_key_rstr( const strm_dict_t *dict, size_t n );
```

Arguments:

dict

A handle to a dictionary object.

n

The 0-based index of the entry whose key is returned.

Library:

libstrm

Description:

The function *strm_dict_key_rstr()* finds the (*n*+1)th entry in the specified dictionary and returns a handle to the entry's key. For example, if *n* is 3, a handle to the the key of the fourth entry is returned. The returned shareable string object handle is owned by the dictionary, and remains valid until the dictionary handle is destroyed.

Returns:

A shareable string object handle to the key of the specified entry if it's found, or a null pointer if it isn't found.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_new()

Create a new handle for an empty dictionary object

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_new( void );
```

Library:

libstrm

Description:

The function *strm_dict_new()* creates a new empty dictionary object and returns a new handle to it.

Note that multiple handles may be represented by identical pointer values; you should not compare handles.

Returns:

A new dictionary object handle.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Dictionary Object API (p. 113)
A dictionary object is a collection of key-value pairs that maps the names of parameters to their values. For **mm-renderer**, you can use the dictionary API to define context, input, and output parameters. Other components can use the same API to manage parameters specific to their purpose.

strm_dict_set()

Modify a dictionary entry (using key-value strings)

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_set( strm_dict_t *dict,
                           const char *key,
                           const char *value );
```

Arguments:

dict

A dictionary object handle.

key

The key of the dictionary entry to add or modify.

value

The value of the dictionary entry to add or modify.

Library:

libstrm

Description:

The function *strm_dict_set()* creates a new dictionary object that is an exact replica of the dictionary object specified by the *dict* argument, except that the entry specified by the *key* and *value* arguments is added or modified. A handle to the new dictionary object is returned. The original dictionary handle is destroyed on success.

Returns:

A handle to the new dictionary object on success, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

Related Links

Dictionary Object API (p. 113)
A dictionary object is a collection of key-value pairs that maps the names of parameters to their values. For **mm-renderer**, you can use the dictionary API to define context, input, and output parameters. Other components can use the same API to manage parameters specific to their purpose.

strm_dict_set_rstr()

Modify a dictionary entry (using key-value shareable string objects)

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_set_rstr( strm_dict_t *dict,
                                 strm_string_t *key,
                                 strm_string_t *value );
```

Arguments:

dict

A dictionary object handle.

key

The key of the dictionary entry to add or modify.

value

The value of the dictionary entry to add or modify.

Library:

libstrm

Description:

The function *strm_dict_set_rstr()* creates a new dictionary object that is an exact replica of the dictionary object specified by the *dict* argument, except that the entry specified by the *key* and *value* arguments is added or modified. A handle to the new dictionary object is returned.

The original dictionary handle, and the handles to the *key* and *value* arguments are destroyed on success.

This function is equivalent to *strm_dict_set()*, except that it may be more efficient if you use clones of the same *key* handle repeatedly.

Returns:

A handle to the new dictionary object on success, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_size()

Return the number of entries in a dictionary

Synopsis:

```
#include <strm.h>

size_t strm_dict_size( const strm_dict_t *dict );
```

Arguments:

dict

A dictionary object handle.

Library:

libstrm

Description:

The function *strm_dict_size()* returns the number of entries in the specified dictionary.

Returns:

The number of entries in the specified dictionary.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_subtract()

Subtract one dictionary from another

Synopsis:

```
#include <strm.h>

strm_dict_t *strm_dict_subtract( strm_dict_t *left, strm_dict_t const *right );
```

Arguments:

left

A handle to the first dictionary object.

right

A handle to the second dictionary object.

Library:

libstrm

Description:

The function *strm_dict_subtract()* creates a replica of the *left* dictionary object and removes all those entries that have matching keys in the *right* object regardless of their value.

Note that the *left* dictionary object handle is consumed by this function, even on a failure, but the *right* is not (unless it's the same handle). If the same handle is passed for both arguments, this function returns an empty dictionary object, without dismantling the dictionary object referenced by the function arguments.

Returns:

A new handle to the resulting dictionary object on success, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_t

Dictionary object type

Synopsis:

```
#include <strm.h>

typedef struct strm_dict strm_dict_t;
```

Description:

The structure **strm_dict_t** is a private data type representing a dictionary object.

Dictionaries cannot be modified; they can only be created and destroyed. For example, the function *strm_dict_set()* takes a dictionary object handle, destroys it, creates a replica of the dictionary object and returns a new handle to it. It's equivalent to calling *strm_string_destroy()* and *strm_string_make()*, except that it may reuse the original object's memory.

Different dictionary object handles may be represented by identical pointers. You should not compare handles. Regardless of how the handles are represented internally, you have to call *strm_dict_destroy()* separately for each handle to properly dispose of any resources associated with it.

Use the following functions to manipulate dictionary objects:

- *strm_dict_clone()*
- *strm_dict_compare()*
- *strm_dict_destroy()*
- *strm_dict_find_index()*
- *strm_dict_find_rstr()*
- *strm_dict_find_value()*
- *strm_dict_index_delete()*
- *strm_dict_key_delete()*
- *strm_dict_key_get()*
- *strm_dict_key_rstr()*
- *strm_dict_new()*
- *strm_dict_set()*
- *strm_dict_set_rstr()*
- *strm_dict_size()*
- *strm_dict_subtract()*
- *strm_dict_value_get()*
- *strm_dict_value_rstr()*

Class:

QNX Neutrino

Related Links

strm_dict_find_index() (p. 117)

Return the index of a dictionary entry

strm_dict_find_rstr() (p. 118)
Find the value of a dictionary entry based on the entry's key (returns a shareable string object handle)

strm_dict_find_value() (p. 119)
Find the value of a dictionary entry based on the entry's key (returns a string)

strm_dict_key_get() (p. 122)
Find the key of a dictionary entry (returns a string)

strm_dict_key_rstr() (p. 123)
Find the key of a dictionary entry (returns a shareable string object handle)

strm_dict_value_get() (p. 133)
Find the value of a dictionary entry based on the entry's index (returns a string)

strm_dict_value_rstr() (p. 134)
Find the value of a dictionary entry based on the entry's index (returns a shareable string object handle)

strm_dict_index_delete() (p. 120)
Delete a dictionary entry (by index)

strm_dict_key_delete() (p. 121)
Delete a dictionary entry (by key)

strm_dict_set() (p. 125)
Modify a dictionary entry (using key-value strings)

strm_dict_set_rstr() (p. 127)
Modify a dictionary entry (using key-value shareable string objects)

strm_dict_value_get()

Find the value of a dictionary entry based on the entry's index (returns a string)

Synopsis:

```
#include <strm.h>

const char *strm_dict_value_get( const strm_dict_t *dict, size_t n );
```

Arguments:

dict

A dictionary object handle.

n

The 0-based index of the entry whose value is returned.

Library:

libstrm

Description:

The function *strm_dict_value_get()* returns the value of the (*n*+1)th entry in the dictionary as a null-terminated string. For example, if *n* is 3, the value of the fourth entry is returned. The returned string is owned by the dictionary object, and remains valid until the dictionary handle is destroyed.

Returns:

The specified key on success, or a null pointer on failure.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_dict_value_rstr()

Find the value of a dictionary entry based on the entry's index (returns a shareable string object handle)

Synopsis:

```
#include <strm.h>

const strm_string_t *strm_dict_value_rstr( const strm_dict_t *dict, size_t n );
```

Arguments:

dict

A handle to a dictionary object.

n

The 0-based index of the entry whose value is returned.

Library:

libstrm

Description:

The function *strm_dict_value_rstr()* finds the (*n*+1)th entry in the dictionary, and returns a handle to shareable string object containing the entry's value. For example, if *n* is 3, a handle to a shareable string object containing the value of the fourth entry is returned. The returned string handle is owned by the dictionary object, and remains valid until the dictionary handle is destroyed.

Returns:

A shareable string object handle to the key of the specified entry if the entry is found, or a null pointer if it isn't found.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_alloc()

Allocate a new shareable string object

Synopsis:

```
#include <strm.h>

char *strm_string_alloc( size_t len, strm_string_t **handle );
```

Arguments:

len

The length of the string to make room for, not including the terminating `'\0'` character.

handle

A pointer to a variable where the new string handle will be stored.

Library:

libstrm

Description:

The function *strm_string_alloc()* allocates a new shareable string object to be filled in by the caller. The caller must put a null-terminated string in the buffer of the new string object before calling any of the functions *strm_string_clone()*, *strm_string_modify()*, or *strm_string_destroy()*, and must not modify the buffer afterwards.

Returns:

A pointer to the first byte of the new string object's string buffer, or a null pointer on error. The new string handle is stored in the variable pointed to by the function argument *handle*.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_clone()

Create a new handle to an existing shareable string object

Synopsis:

```
#include <strm.h>

strm_string_t *strm_string_clone( const strm_string_t *sstr );
```

Arguments:

sstr

A handle to a shareable string object.

Library:

libstrm

Description:

The function *strm_string_clone()* creates a new string handle to the shareable string object referenced by the argument *sstr*.

Returns:

A new handle to the shareable string handle on success, or a null pointer on failure (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_destroy()

Destroy a string handle

Synopsis:

```
#include <strm.h>

int strm_string_destroy( strm_string_t *sstr );
```

Arguments:

sstr

A handle to a shareable string object.

Library:

libstrm

Description:

The function *strm_string_destroy()* destroys the specified string handle, and frees the memory allocated for the string object if the specified handle is the last one referencing it.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_get()

Return a pointer to the first character of the string

Synopsis:

```
#include <strm.h>

const char *strm_string_get( const strm_string_t *sstr );
```

Arguments:

sstr

A handle to a shareable string object.

Library:

libstrm

Description:

The function *strm_string_get()* returns a pointer to the first character of the string in the shareable string object referenced by the *sstr* argument. This string should be considered read only; your application shouldn't attempt to modify it.

Returns:

A pointer to the first character of the string in the shareable string object referenced by the *sstr* argument.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_make()

Create a new shareable string object

Synopsis:

```
#include <strm.h>

strm_string_t *strm_string_make( const char *cstring );
```

Arguments:

cstring

A pointer to a null-terminated string.

Library:

libstrm

Description:

The function *strm_string_make()* creates a new shareable string object, populating it with the string passed in the *cstring* argument, and returning a handle to the new string object.

Returns:

A handle to the new string object.

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_modify()

Modify a shareable string object (destroying the existing handle)

Synopsis:

```
#include <strm.h>

strm_string_t *strm_string_modify( strm_string_t *sstr, const char *cstring );
```

Arguments:

sstr

A handle to a shareable string object.

cstring

A pointer to a null-terminated string.

Library:

libstrm

Description:

The function *strm_string_modify()* creates a new shareable string object from the string passed in the *cstring* argument, and returns a new handle to the new string object. Calling *strm_string_modify()* is equivalent to calling *strm_string_destroy()* and *strm_string_make()*, except that it may reuse the original object's memory. The shareable string object handle passed in the *sstr* argument is consumed, even on failure.

Returns:

A new handle to the new shareable string object on success, or a null pointer on failure (*errno* is set).

Classification:

QNX Neutrino

Safety:	
Interrupt handler	No
Signal handler	No
Thread	Yes

strm_string_t

Shareable string type

Synopsis:

```
#include <strm.h>

typedef struct strm_string strm_string_t;
```

Description:

The structure **strm_string_t** is a private data type representing a shareable string.

Shareable strings cannot be modified; they can only be created and destroyed. For example, the function *strm_string_modify()* takes a shareable string object handle, destroys the object, creates a new shareable string object and returns a new handle to it. It is equivalent to calling *strm_string_destroy()* and *strm_string_make()*, except that it may reuse the original object's memory.

Different shareable string object handles may be represented by identical pointers. You should not compare handles. Regardless of how the handles are represented internally, you have to call *strm_string_destroy()* separately for each handle to properly dispose of any resources associated with it.

Use the following functions to manipulate shareable strings:

- *strm_string_alloc()*
- *strm_string_clone()*
- *strm_string_destroy()*
- *strm_string_get()*
- *strm_string_make()*
- *strm_string_modify()*

Class:

QNX Neutrino

Index

C

connecting to `mm-renderer` *19*
contexts *14–17*, *24*, *35*
    closing handles *24*
    creating and destroying *24*
    description *14*
    input *16*
    orphan contexts *24*
    output *15*
    plugins *17*
    primary handle *24*
    secondary handles *24*
    state information *35*

D

dictionaries *114–125*, *127*, *129–130*, *133–134*
    cloning *114*
    comparing *115*
    creating *124*
    deleting entries *120–121*
    destroying *116*
    determining size of *129*
    finding entries *117*
    finding keys *122–123*
    finding values *118–119*, *133–134*
    modifying entries *125*, *127*
    subtracting *130*
Dictionary Object API *113*, *131*, *141*
    dictionary object type *131*
    introduction *113*
    shareable string type *141*
disconnecting from `mm-renderer` *19*

M

`mm-renderer` *11–14*, *19–20*, *22–24*, *26–28*, *33*, *35–36*, *39–40*, *42*, *44*, *54–56*, *61*, *63*, *70*, *78*, *82–83*, *95*, *105*
    API *39–40*, *42*, *44*, *54–56*, *61*, *63*, *70*, *78*, *82–83*, *95*, *105*
        checking for errors *55*
        configuring inputs *83*
        configuring outputs *95*
        connecting and disconnecting *40*
`mm-renderer` *(continued)*
    API *(continued)*
        connection handle type *42*
        context handle type *54*
        context states *82*
        controlling playback *105*
        error codes *56*
        error information structure *61*
        event information structure *70*
        event types *78*
        events *63*
        header file locations *39*
        introduction *39*
        managing contexts *44*
    architecture *13*
    capabilities *11*
    command line *22–23*
        description *23*
        options *22*
        syntax *22*
    configuration file *20*
    connecting to *19*
    contexts, *See* contexts
    detecting changes in play state *36*
    detecting warnings and errors *36*
    disconnecting from *19*
    managing video windows *28*
    operations for connected clients *19*
    parameters, *See* parameters
    play speed *27*
    playing media *27*
    PPS objects, *See* PPS objects
    recording audio data *33*
    rendering video *28*
    seeking to positions *28*
    starting *20*
    supported media categories *12*
    working with contexts *24*
multimedia renderer, *See* `mm-renderer`

P

parameters *26*
    defining *26*

playback *27–28*, *36*
- how to play media *27*
- play state, warnings, and errors *36*
- play states *27*
- seeking *28*
- setting speed of *27*

playlists *12*
- definition *12*
- supported types *12*

playlists and playlist windows *37*
PPS objects *35–38*
- context state *35*
- input metadata *37*
- play state, warnings, and errors *36*
- playlist windows *37*
- summary *35*
- supported file and MIME types *38*

PPS service *11*

S

Screen *28*
- using to manipulate video output *28*

shareable strings *135–140*
- allocating *135*
- cloning *136*
- create from strings *139*
- destroying *137*
- getting string pointers *138*
- modifying *140*

starting `mm-renderer` *20*
supported file and MIME types *38*

T

Technical support *10*
tracks *12*
- definition *12*
- supported sources *12*

trick play *110*
Typographical conventions *8*