Metadata Provider Library Reference

©2013–2015, QNX Software Systems Limited, a subsidiary of BlackBerry Limited. All rights reserved.

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/

QNX, QNX CAR, Momentics, Neutrino, and Aviage are trademarks of BlackBerry Limited, which are registered and/or used in certain jurisdictions, and used under license by QNX Software Systems Limited. All other trademarks belong to their respective owners.

Electronic edition published: March 31, 2015

Contents

About This Reference

The *Metadata Provider Library Reference* is aimed at developers who want to write applications that use the **libmd** library to extract metadata from media files on attached devices. This metadata lets applications display track information and artwork so users can quickly browse device filesystems and search media libraries.

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

To find out about:	Go to:
The purpose and capabilities of **libmd**	*Metadata Provider Overview* (p. 9)
The list of included Metadata Providers (MDPs)	*Included MDPs* (p. 12)
The **libmd** configuration file, which lists the plugins and their preferential order	*Configuration file* (p. 18)
Using the Metadata Provider Library API to manage metadata-extraction sessions and retrieve metadata from media files	*Metadata Provider API* (p. 21)

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.

Copyright © 2015, QNX Software Systems Limited

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.

Copyright © 2015, QNX Software Systems Limited

Chapter 1
Metadata Provider Overview

The metadata provider library, **libmd**, extracts metadata from media files on attached devices to provide client applications with up-to-date information on the media content available for browsing or playback.

Metadata is information that describes media files. This information can include details such as the artist name, album title, or year of creation (for a track), as well as playback details such as track runtime or picture dimensions. Clients use metadata to:

- display details of the currently playing track to users
- provide the names and other file information of selected tracks to media browsers
- display album artwork to users to enhance their media experience
- generate cover flows so users can visually browse albums and tracks

Clients make requests of **libmd** to extract specific sets of metadata fields from individual media files. With this design, clients can retrieve the exact metadata they need at precise times so they can optimize performance and the user experience. For example, suppose the user selects a track in their media browser. The application that provides media information to the browser can extract the track's creation information fields (which are small and fast to retrieve) but not its embedded artwork images (which can be large and slow to process). This strategy increases the browser's responsiveness.

Consider a cover flow application that allows users to visually browse their song collection and begin playback by selecting an album image. The application can extract the cover art when generating the flow of albums and then extract the artist name, year of release, and other album information when the user selects an album image. This “load-on-demand” philosophy supports a good user experience by ensuring the exact information—whether images or text—becomes available as soon as the user needs it.

The multimedia synchronizer service, **mm-sync**, uses **libmd** to extract media metadata so it can upload that metadata to QDB databases. But updating databases requires indexing most or all files on a mediastore, so you might not want to rely on **mm-sync** for obtaining metadata and instead use **libmd**. Retrieving metadata through **libmd** lets you prioritize metadata extraction by reading metadata from one file or a select group of files.

Architecture of libmd

The **libmd** library uses a plugin architecture in which independent plugins support different sets of metadata fields. When a client requests metadata, the library extracts it using one or more plugins and then returns the set of filled-in metadata fields to the client.

The library is implemented in three layers:

Data processing

This layer:

- stores and updates the plugin ratings for metadata fields from specific media file types
- collates (i.e., combines and orders) the metadata field values returned from plugins

Plugin management

This layer:

- parses the configuration file to learn which library files implement the plugins and to read the preference order for various file types
- loads, initializes, and unloads plugins

Plugins

This layer consists of many Metadata Provider (MDP) plugins, each of which:

- manages communication sessions for responding to metadata requests and for reporting errors
- rates itself on its ability to retrieve the requested metadata fields
- retrieves metadata by extracting media information from the named item (media file)

This design lets **libmd** offer a common, high-level interface for extracting metadata from many file types on many device types. Clients need to name only a media file and the metadata fields they want and **libmd** then invokes the necessary MDP plugins to read the metadata and returns the extracted metadata fields to the client.

Each MDP fills in as many fields as it can. The order that **libmd** uses to invoke the MDPs depends on the plugin preferences stated in the *configuration file* (p. 18). The preferential order for plugins can vary from one file type to another.

Supported file types

The file types and their associated URL prefixes supported by **libmd** are:

File type	URL prefix
CDDA track	**cdda:**
POSIX file	**file:**
iPod media file	**ipod:**
HTTP stream	**http:**

Copyright © 2015, QNX Software Systems Limited

File type	URL prefix
RTSP stream	**rtsp:**
File on an MTP device	**mtp:**

If no URL prefix is given, the POSIX file type is assumed (e.g., a URL of **/fs/usb0/one.mp3** is equivalent to **file:/fs/usb0/one.mp3**).

Supported device types

MDPs hide the details of the media interface used for reading metadata so clients can extract it through different network protocols from a variety of hardware. Clients can read metadata from the following device types:

- USB sticks, SD cards, or any storage devices with block filesystems
- iPods
- audio CDs
- MTP devices
- media streams from external sources (e.g., HTTP servers)

The plugin-based architecture makes it possible for future releases of **libmd** to support additional file and device types. The **libmd** library could add new MDPs to provide more sources of metadata while clients continue to use the same commands to extract it.

Metadata providers

Metdata providers (MDPs) are **libmd** plugins that do the actual metadata extraction from media files. MDPs tell the data-processing layer of **libmd** which metadata fields (types) they can extract. When requested, MDPs read as many of the metadata fields listed by the client as possible from the specified media item.

MDP ratings

To handle a client request for metadata, **libmd** queries all loaded MDPs for their ratings on the metadata fields listed in the request. Each MDP keeps an internal map of the fields it can extract from media files. This map contains the field names (i.e., metadata types) and other information such as which collation method to use for handling multiple values for a given field. MDPs consult this map to generate lists of field-specific Boolean ratings (1 means the plugin can extract the field, 0 means it can't) and then return these ratings to the data-processing layer.

When selecting an MDP plugin as the metadata source, **libmd** considers only MDPs that gave a rating of 1 for at least one metadata field, which means they can extract some or all of the requested information. To pick an MDP within the set of MDPs rated 1 for some field, **libmd** examines the plugin order for the file type of the media item named in the request. This plugin order is read from the configuration file during initialization (see *Configuration file* (p. 18)).

Metadata extraction

When **libmd** asks an MDP to retrieve metadata, the selected MDP parses the request data to obtain either a fully qualified path to the item (media file) or some other information referencing the item (e.g., a track's unique ID (UID) on iPods). Next, the MDP uses POSIX system calls or system libraries to browse the device's directories and files and to read its file information to generate metadata. For instance, the CDDA plugin calls *devctl()* to issue device commands to CDs mounted in the local filesystem. These commands include reading the CD-Text data, which contains album metadata.

The MDP stores the information read from the device in metadata strings and returns these strings along with the number of metadata types (fields) for which metadata was found to the data-processing layer of **libmd**. If the number of types found is less than the number requested by the client, **libmd** picks another MDP to get metadata for the remaining fields. The **libmd** library continues invoking MDPs until all requested metadata fields have been filled in or until it exhausts all MDPs.

Included MDPs

Different MDPs support different file types and metadata fields. When requesting metadata fields, you must state the metadata categories and the individual attributes that you want to retrieve; see *mmmd_get()* (p. 30) for more details.

The **libmd** library combines a category (the prefix) with each of the listed attributes (the suffixes) to form the full names of the metadata fields. You can state as many categories and attributes as you need but you should be aware of which fields are supported by which MDPs.

The MDPs shipped with **libmd** and the file types and metadata that they support are as follows:

Copyright © 2015, QNX Software Systems Limited

MDP	Files	Metadata categories	Attributes
CDDA	CD audio tracks	`md_title`	`album`, `artist`, `genre`, `name`, `composer`, `track`, `bitrate`, `samplerate`, `duration`, `format`
Exif	POSIX files on mass storage devices (e.g., USB sticks)	`md_title`	`width`, `height`, `date_time_original`, `shutter_speed`, `fnumber`, `iso_speed_ratings`, `focal_length`, `orientation`, `description`, `latitude`, `longitude`, `keywords`
Extart	External artwork such as cover images for albums and thumbnail graphics for tracks	`md_title`	`art` (see 1)
		`md_artwork`	`image` (see 2), `count`, `size`, `urls`
IMG	Image files	`md_title`	`width`, `height`, `color_depth`
iPod	iPod media files	`md_title`	`art` (see 1)
		`md_artwork`	`image` (see 2), `mimetype`, `width`, `height`, `size`, `count`
MediaFS	Files on MTP devices	`md_title`	`name`, `artist`, `album`, `composer`, `genre`, `year`, `duration`, `comment`, `protected`, `track`, `art` (see 1)
		`md_artwork`	`image` (see 2), `width`, `height`, `size`, `mimetype`, `count`
MMF	MMF files accessible from either a network source (e.g., an HTTP server) or a POSIX device	`md_title`	`name`, `artist`, `album`, `albumartist`, `composer`, `genre`, `comment`, `duration`, `track`, `disc`, `year`, `seekable`, `pausable`, `samplerate`, `bitrate`, `protected`, `mediatype` (see 3), `width`, `height`, `art` (see 1), `compilation`, `rating`
		`md_video`	`width`, `height`, `pixel_width`, `pixel_height`, `frame`, `fourcc`
		`md_audio`	`fourcc`
		`md_artwork`	`image` (see 2), `description`, `type`, `mimetype`, `count`, `size`

1 This field is an alias for `md_artwork::image`, which is the preferred way of requesting image data.

2 The `image` field must include a `file` option that names the path to write the image data. This can be a POSIX path in the local filesystem or a special value to request an image file reference instead of the image data (for more information, see "*Metadata provider constants* (p. 22)"). The field can also specify an index to select one image from many within the given path (for details, see "*Extracting artwork* (p. 15)").

3 The value returned for the `md_title::mediatype` field is in decimal but should be converted to hexadecimal for readability. For the mapping of hexadecimal values to media types, see the *MediaFormat_t* data structure in the *Addon Interface Library Reference*.

Metadata-extraction sessions

To extract metadata with **libmd**, a client must establish a communication session with the library before it can issue commands to read metadata from media files stored on an attached device.

To establish a communication session (or *metadata-extraction session*) with **libmd**, the client must name a mediastore (device) to extract the metadata from. If desired, the client can then set session parameters to influence the behavior of MDPs. These parameters can be set only once, so the client should set them just after opening a session but before extracting any metadata.

The client can use any open session to send requests to **libmd** to extract metadata from individual items (media files). In each metadata request, the client must supply the item's path or some device-specific information identifying the item (e.g., a UID on iPods) and must list the desired metadata fields. The client can also request a maximum number of “matches” (i.e., responses returned by different plugins) for metadata fields. Retrieving multiple values for metadata fields lets a client pick the set of values that provide the user with the most complete and accurate media information possible.

Concurrent sessions

Clients can open and extract metadata from as many concurrent **libmd** sessions as they like. This design lets applications display media information for multiple devices to users. We recommend a limit of one session per mediastore to avoid redundant reads of metadata from the same files.

Obtaining error information

While a session is active, the client can obtain information about the last error that occurred for that session by calling *mmmd_error_info()* (p. 25). This function returns error information, including the numeric error code, a string summarizing the error, and an error message. We recommend that your client code check the return values of all API calls. If any value indicates an error, the client can retrieve the error information and use it to help recover.

Copyright © 2015, QNX Software Systems Limited

Extracting artwork

Media content often has artwork. For example, audio and video files can have embedded images, thumbnail graphics, and album artwork residing in their folder as well. The **libmd** library can read image metadata, allowing clients to discover artwork on connected mediastores and to retrieve images so they can display them during playback and enhance the user experience.

To extract artwork, you must:

1. Enable and configure artwork plugins.

   The **libmd** library extracts artwork with these plugins:

   Extart

   Searches folders on connected mediastores to find artwork associated with media files; retrieves image data, URLs, and other properties

   iPod

   Reads image data, file information, and display dimensions from specified artwork files on Apple devices

   MediaFS

   Reads image data, file information, and display dimensions from specified artwork files on MTP devices

   MMF

   Reads image data, descriptions, and properties of artwork accessed through network sources and POSIX devices

   In the default configuration file, the MMF plugin is the only artwork-supporting plugin enabled. To enable another plugin, uncomment the `[plugin]` line that starts its configuration section and the `dll` setting on the next line. The Extart plugin has additional settings that you can uncomment to control some aspects of artwork extraction, such as the maximum number of images to read. These settings are explained in "*MDP Settings* (p. 20)".

2. Add artwork plugins to lists of preferred plugins.

   For all file types for which you want to retrieve artwork, you must name at least one artwork-supporting plugin in their lists of preferred plugins. Suppose you want to read artwork for media files found in POSIX filesystems and on MTP devices. You can modify the `[typeratings]` section in the *configuration file* (p. 18) as follows:

   ```
   [typeratings]
   file=extart,mmf,img
   mtp=extart,mediafs
   ```

   This tells **libmd** to use the Extart plugin for reading metadata when given a URL starting with **file:** or **mtp:**, or with no prefix and containing only a POSIX path (which is equivalent to using the **file:** prefix). In both cases, **libmd** tries to read the metadata with Extart. If it can't retrieve some fields with this plugin, the library may try the other listed plugins, depending on the parameters passed in by the client when requesting metadata.

3. Retrieve artwork through the **libmd** API.

   The API call sequence for retrieving artwork is the same as that for reading other metadata. See the *Metadata Provider API* (p. 21) chapter for a summary of the commands used to read metadata. The key point to remember when extracting artwork is that not all information gets copied into the memory space referenced by the *md* parameter in the *mmmd_get()* (p. 30) call. Image details such as size or URL are stored in this library-allocated memory but the image data (which is much larger) is copied to the path specified in the `md_artwork::image` metadata type, which is named in the *types* parameter.

   Suppose the client configures the Extart plugin to read POSIX files and then makes this API call:

```
mmmd_get( hdl, "/mediastore/1.mp3",
          md_artwork::image?file=/tmp/img1.jpg", NULL,
          0, &md );
```

   Although the second argument defines a file path, the Extart plugin looks in the enclosing folder (**/mediastore** in this case) for artwork images. Because no image index is given in this example, the library writes the first image found that matches the search pattern defined by the `regex`# configuration settings into **/tmp/img1.jpg**.

   For any artwork-supporting plugin, you can define the `index` parameter to extract a particular image. To know if more than one image is present in a given folder, you must first read one of the following types in an *mmmd_get()* call:

   `md_artwork::count`

   Indicates the number of images in the folder

   `md_artwork::size`

   Stores the sizes of all images found, in a comma-separated list of values

   `md_artwork::urls`

   (Supported only by Extart)

   Stores the URLs of the images, in a comma-separated list of values. Some clients may not want to copy images and instead read and display them directly from devices. In this case, they can request the `md_artwork::urls` field, read the URLs within this field, and then specify these URLs in POSIX system calls to access images individually.

   Suppose you learn of multiple artwork images associated with an audio track and you want to display these images at specific time offsets (e.g., 0s, 30s, 60s, and so on) during playback. At the appropriate times, you can request new images by specifying their indexes in calls like this:

```
mmmd_get( hdl, "/mediastore/1.mp3",
          md_artwork::image?file=/tmp/img1.jpg,index=1", NULL,
          0, &md );
```

   This command retrieves the image at index 1 (i.e., the second image in the **/mediastore** folder). You could retrieve this image if you want to refresh the display after, say, 30 seconds of playback.

Copyright © 2015, QNX Software Systems Limited

Chapter 2
Configuring Metadata Providers

You can configure metadata providers (MDPs) in two ways: in the configuration file to define initial settings and through the **libmd** API to define settings for individual metadata-extraction sessions.

During startup, **libmd** reads its configuration file and loads each listed MDP. After an MDP loads successfully, **libmd** initializes it with any settings listed in the configuration file. These settings apply to the MDP throughout the client application's lifetime.

When **libmd** has finished its setup, your client can establish metadata-extraction sessions and assign parameters to those sessions to influence how MDPs retrieve metadata.

To assign parameters to active sessions (*dynamic parameters*), the client must call *mmmd_session_params_set()* (p. 38) while providing the session handle and the list of parameters. Parameters defined in this manner apply only to the session referred to in the API call. Once set, they can't be changed or unset.

Currently, only the MMF MDP examines dynamic parameters, which it uses to configure the streamers for reading files from HTTP servers. Whether this plugin is used in metadata extraction depends on the type of the media item being read and the plugin preferences stated in the configuration file. When **libmd** uses other MDPs to read metadata, dynamic parameters have no effect. You should therefore set these parameters only when you plan to extract metadata from HTTP servers.

The MDP settings recognized by **libmd** when parsing the configuration file (*static parameters*) differ from those you can assign to an active metadata-extraction session. See the default configuration file for the supported static parameters. For information on the dynamic parameters recognized by MMF, see *mmmd_session_params_set()* (p. 38).

Configuration file

The **libmd** configuration file lists the preferential plugin order, the library files implementing the plugins, and other configuration settings.

The **libmd** library is shipped with a default configuration file. You can modify this included file or create your own. You can also override the default path that the library looks in for the configuration file, when calling *mmmd_init()* (p. 34). If you pass in a configuration path of NULL, the library searches the path given in the MM_MD_CONFIG environment variable or if this variable isn't defined, the default path of **/etc/mm/mm-md.conf**.

Redefining MM_MD_CONFIG lets you use a different configuration file as the default. This is useful when launching applications such as **mm-sync** that use **libmd** but don't allow you to set the configuration path.

In any configuration file, each section that defines settings for an individual plugin (or MDP) must begin with a line like this:

```
[plugin]
```

The settings are listed on the lines that follow, one per line. A setting is specified by stating a field name, followed by an equal sign (`=`), followed by the field value. For example, the following line enables “lazy load filters” for MMF:

```
lazyloadfilters=1
```

You can add comments in the file by starting lines with the number sign (`#`).

A `dll` setting is required in every plugin section. This setting names the library file implementing the MDP plugin. To support a good user experience, your configuration file should define at least all the MDPs needed to extract any metadata field used by your client applications. Most likely, you'll have to provide more than one plugin section in your configuration because most MDPs don't support every metadata field.

The section defining the preferential plugin order begins with a line of the form:

```
[typeratings]
```

The lines that follow list the MDP preferences for specific file types. Each line contains a URL prefix that represents a file type, followed by the MDPs to use for metadata extraction, from most to least preferred. Suppose you want to inform **libmd** of your plugin preferences for POSIX files, whose URLs have either a **file** prefix or no prefix at all. If you want to use the **MMF** MDP first, then the **Exif** MDP if some metadata fields can't be retrieved by this first MDP, and then the **Img** MDP if some fields still can't be retrieved, enter the following line:

```
file=mmf,exif,img
```

Copyright © 2015, QNX Software Systems Limited

Default configuration file

The contents of the default configuration file look like this:

```
# libmd config file

[plugin]
dll=mm-mdp-mmf.so
lazyloadfilters=1

[plugin]
dll=mm-mdp-cdda.so

#[plugin]
#dll=mm-mdp-exif.so

#[plugin]
#dll=mm-mdp-img.so

#[plugin]
#dll=mm-mdp-ipod.so

#[plugin]
#dll=mm-mdp-extart.so
#ignore_case=true
#max_search=100
#max_cache_entries=0

# All regular expressions following the first instance must have
# a unique suffix appended to them (e.g., regex, regex1, regex2).
#regex=album\.jp[e]?g
#regex1=folder\.jp[e]?g

#[plugin]
#dll=mm-mdp-mediafs.so

[typeratings]
file=mmf
#file=mmf,exif,img
http=mmf
cdda=cdda
rtsp=mmf
#ipod=ipod
#mtp=mediafs
```

Some MDP settings are commented out in the default file; to enable any of these settings, simply uncomment its line.

MDP Settings

You can configure these MDP settings to further control metadata extraction:

MDP	Setting	Description
Extart	`ignore_case`	Use case-insensitive matching of filenames; this is usually desired
	`max_cache_entries`	Limit the number of folders cached after searching; this helps limit memory usage
	`max_search`	Limit the number of files to check in a folder when looking for external artwork. Defining this setting lets you keep the search time within a reasonable limit. For example, if a folder contains 10 000 files but `max_search` is 100, only the first 100 files will be checked for artwork.
	`regex[#]`	Define POSIX regular expression (regex) patterns for matching names of artwork files. You can define multiple fields; the first can be named `regex` but the remaining fields must contain unique suffixes (e.g., `regex1`, `regex2`).
MMF	`lazyloadfilters`	Don't initialize MMF until the first request for metadata; this saves on startup time

Copyright © 2015, QNX Software Systems Limited

Chapter 3
Metadata Provider API

The Metadata Provider API exposes the constants, data types (including enumerations), and functions that client applications can use to initialize the **libmd** library, create metadata-extraction sessions, and submit metadata retrieval requests.

The first action any client must perform with **libmd** is to initialize the library by calling *mmmd_init()* (p. 34) while supplying the path of the configuration file, which lists the metadata providers (MDPs) to load.

Before it can extract any metadata, the client must open a *metadata-extraction session* by calling *mmmd_session_open()* (p. 37) while providing the name of the mediastore (device) to read metadata from.

The client can then request specific metadata fields from specific items (media files) by calling *mmmd_get()* (p. 30). The client can ask for a maximum number of *matches* (i.e., responses from different MDPs). Retrieving multiple matches lets the client pick the set of metadata values that provide the most complete and accurate media information possible.

When it's finished retrieving metadata, the client can close the corresponding session by calling *mmmd_session_close()* (p. 36). When it's finished using **libmd** altogether (e.g., during shutdown), the client must call *mmmd_terminate()* (p. 39) to clean up the resources used by the library.

Metadata provider constants

Constants for requesting cover art references instead of artwork data

Synopsis:

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

```
#define MD_COVERART_BYREF "BYREF"
```

This constant provides a keyword for returning cover art by reference. If requested, MDPs won't write the artwork data to a file. You can use this constant when specifying the `image` attribute in the list of metadata types to retrieve, for example: `md_artwork::image?file=BYREF`.

Library:

libmd

Copyright © 2015, QNX Software Systems Limited

mmmd_errcode_t

Error codes

Synopsis:

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

typedef enum mmmd_errcode {
    MMMD_ERR_NONE = 0,
    MMMD_ERR_OTHER,
    MMMD_ERR_NO_MDPS,
    MMMD_ERR_NOT_SUPPORTED,
    MMMD_ERR_MALFORMED_REQUEST,
    MMMD_ERR_NO_PARSERS,
    MMMD_ERR_CALLDEPTH_EXCEEDED,
    MMMD_ERR_NO_MEMORY,
    MMMD_ERR_CANT_OPEN_FILE,
    MMMD_ERR_CANT_READ_FILE,
    MMMD_ERR_CANT_RECONFIGURE,
} mmmd_errcode_t;
```

Data:

MMMD_ERR_NONE

No error occurred.

MMMD_ERR_OTHER

An error not listed here occurred.

MMMD_ERR_NO_MDPS

No metadata plugins are loaded.

MMMD_ERR_NOT_SUPPORTED

The request isn't supported.

MMMD_ERR_MALFORMED_REQUEST

The request isn't properly formed.

MMMD_ERR_NO_PARSERS

No parsers were found for the request.

MMMD_ERR_CALLDEPTH_EXCEEDED

The derived metadata call depth was exceeded (presently not applicable).

MMMD_ERR_NO_MEMORY

No memory is available.

MMMD_ERR_CANT_OPEN_FILE

The file couldn't be opened.

MMMD_ERR_CANT_READ_FILE

The file couldn't be read.

MMMD_ERR_CANT_RECONFIGURE

The configuration was already set (presently not applicable).

Library:

libmd

Description:

The **mmmd_errcode_t** enumeration defines the error codes that can be returned by **libmd** API functions.

Copyright © 2015, QNX Software Systems Limited

mmmd_error_info()

Get information about the last error in a session

Synopsis:

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

const mmmd_error_info_t* mmmd_error_info( mmmd_hdl_t *hdl )
```

Arguments:

hdl

The handle of the session whose error information is being retrieved.

Library:

libmd

Returns:

A pointer to an **mmmd_error_info_t** (p. 26) structure storing the error information.

mmmd_error_info_t

Information about the last session error

Synopsis:

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

typedef struct mmmd_error_info {
    mmmd_errcode_t code;
    int64_t extended_code;
    char extended_type[16];
    char extended_msg[256];
} mmmd_error_info_t;
```

Data:

mmmd_errcode_t code

The numeric error code.

int64_t extended_code

The numeric extended error code.

char extended_type

The extended error type, as a string.

char extended_msg

An extended error message.

Library:

libmd

Description:

The **mmmd_error_info_t** structure describes errors that occurred during a metadata-extraction session.

Copyright © 2015, QNX Software Systems Limited

mmmd_error_str()

Get a phrase describing the specified error code

Synopsis:

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

const char* mmmd_error_str( mmmd_errcode_t errcode )
```

Arguments:

errcode

An **mmmd_errcode_t** (p. 23) constant representing the error that you want a descriptive phrase for.

Library:

libmd

Returns:

A pointer to a string containing the error phrase (this value is always non-null).

mmmd_flags_set()

Set control flags for the library logs

Synopsis:

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

mmmd_flags_t mmmd_flags_set( mmmd_flags_t new_flags )
```

Arguments:

new_flags

An **mmmd_flags_t** (p. 29) value with the new flag setting for the library logs.

Library:

libmd

Returns:

The old flag setting, as an **mmmd_flags_t** value.

Copyright © 2015, QNX Software Systems Limited

mmmd_flags_t

Flags for controlling library logs

Synopsis:

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

typedef enum {
    MMMD_FLAG_EMIT_TIMING_LOGS = 0x01
} mmmd_flags_t;
```

Data:

MMMD_FLAG_EMIT_TIMING_LOGS

Tells the library to emit timing logs.

Library:

libmd

Description:

The **mmmd_flags_t** enumeration defines constants for controlling logs for the library.

mmmd_get()

Get metadata fields from a media item

Synopsis:

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

int mmmd_get( mmmd_hdl_t *hdl,
              const char *item,
              const char *types,
              const char *source,
              uint32_t count,
              char **md )
```

Arguments:

hdl

The handle of the session associated with the mediastore from which metadata is being read.

item

A URL or an absolute path to the file containing the metadata being read. For URLs, the prefix depends on the type of file being read (see "*Supported file types* (p. 10)" for details).

types

A string storing the requested metadata types (fields) as a series of *group-attributes* listings. Here, *group* refers to the metadata category (e.g., `title`) while *attributes* refers to the list of requested attributes (e.g., `artist`, `album`).

Each group-attributes listing must be followed by a line-break character (`\n`). Within a listing, the group and the attributes must be separated by the `::` delimiter, while individual attributes must be separated by commas, as shown in this example with two listings:

`md_title::name,artist,album\nmd_video::width,height`

This syntactic grouping of metadata types makes it easy to request multiple related fields.

source

A string specifying the metadata source (i.e., the MDP to use). Currently, this feature isn't supported so this argument must be NULL to indicate that all sources can be used.

count

The number of desired matches (i.e., responses from MDPs).

If *count* is 0, all responses are collated to return the highest-rated response (see the *Description* (p. 31) for an explanation).

If *count* is nonzero, the number of responses returned is less than or equal to *count*, starting with the highest-rated response. No collation is performed.

Copyright © 2015, QNX Software Systems Limited

md

A pointer to a string reference to the buffer storing the response. The library allocates the buffer memory, writes the metadata in this memory, and sets the string reference but the caller owns the memory and hence, is reponsible for freeing it later.

Examples of the formatting and typical contents of the response buffer are given in the *Description* (p. 31).

Library:

libmd

Description:

This function gets the specified metadata fields from the specified item. The *types* string must state the requested fields as group-attributes listings, as explained in the *Arguments* (p. 30).

Because different MDPs support different fields, **libmd** uses as many MDPs as necessary to extract metadata for all the fields listed in *types*. The order that **libmd** uses to invoke the MDPs is the plugin preference order for the file type indicated by the URL or path in *item*. This file type-based preference order is stated in the configuration file.

For the lists of fields supported by different MDPs, see "*Included MDPs* (p. 12)".

The metadata pointer (*md*) should be deallocated using *free()* when the metadata is no longer needed. The **libmd** library sets this pointer to a valid, non-null value only if the return value is greater than 0, meaning metadata was found.

Examples:

Retrieving multiple responses

Setting *count* to a value greater than 0 allows you to retrieve multiple matches (responses) for metadata fields. Your client code can then choose the set of responses that provides the user with the most accurate and complete metadata possible. The number of responses returned is less than *count* if the number of MDPs supporting any of the requested fields is also less than *count*. A nonzero value for this argument simply limits the number of responses that can be returned.

Suppose a client sets *count* to 3 and requests the `md_title_artist` and `md_title_orientation` fields from a POSIX file while the MDP preference order for POSIX files is `mmf, mediafs, exif`. The **MMF** and **MediaFS** MDPs support the first field but not the second; the **Exif** MDP supports the second field but not the first. The **libmd** library then stores a pointer in *md* that references the following string:

```
md_src_name::mmf\nmd_src_rating::0\nmd_title_artist::some_artist\n
md_src_name::mediafs\nmd_src_rating::1\nmd_title_artist::some_artist\n
md_src_name::exif\nmd_src_rating::2\nmd_title_orientation::landscape\0
```

The name and rating of the MDP that produced the metadata are placed in front of every metadata field. Ratings are offsets in the zero-based list of preferred MDPs, so 0 indicates the first plugin listed, 1 indicates the second listed, and so on. The metadata is represented as a name-value pair and placed after the MDP name and rating.

Retrieving the highest-rated responses

Setting *count* to 0 makes **libmd** collate the responses from many MDPs into one result set to produce the highest-rated response, which is the set of metadata field values obtained from the MDPs listed earliest in the plugin preference order.

Suppose a client sets *count* to 0 and requests the `md_title_width`, `md_title_height`, and `md_title_orientation` fields from a POSIX file while the MDP preference order is the same as listed in the last example. The **MMF** and **MediaFS** MDPs support the first two fields but not the last; only the **Exif** MDP supports the last field. The **libmd** library then sets *md* to reference the following string:

```
md_title_width::response_from_MMF\nmd_title_height::response_from_MMF\n
md_title_orientation::response_from_Exif
```

Because **MMF** is rated ahead of **MediaFS**, this first MDP's values for `md_title_width` and `md_title_height` are returned. Neither **MMF** nor **MediaFS** supports `md_title_orientation`, so the value from **Exif** for this last field is returned. Note that the MDP names and ratings aren't shown for individual fields in this case because the responses come from potentially many MDPs.

Returns:

>0

The number of responses, when successful.

0

No metadata was retrieved but no errors occurred.

-1

An error occurred (call *mmmd_error_info()* (p. 25) for details).

Copyright © 2015, QNX Software Systems Limited

mmmd_hdl_t

Session handle type

Synopsis:

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

typedef struct mmmd_hdl mmmd_hdl_t;
```

Library:

libmd

Description:

The **mmmd_hdl_t** structure is a private data type representing a session handle.

mmmd_init()

Initialize the library

Synopsis:

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

int mmmd_init( const char *config )
```

Arguments:

config

The path to the configuration of the library. Setting this argument allows you to use a nondefault configuration file. When given a NULL path, the library searches the path given in the MM_MD_CONFIG environment variable or if this variable isn't defined, the default path of **/etc/mm/mm-md.conf**.

Library:

libmd

Description:

This function initializes the library. You must call this function before any other **libmd** function to initialize the library before using it. This function loads any metadata providers (MDPs) listed in the configuration file into the library. The default path for the configuration file is **/etc/mm/mm-md.conf** but this path can be overridden, as explained in the *config* argument.

The plugin entries in the configuration file must contain `dll` settings that provide filenames matching the plugin names. All other plugin settings are ignored by the data processing and plugin management layers but may be used by the plugins themselves during metadata extraction.

Returns:

0

Success.

>0

An error occurred (*errno* is set).

Copyright © 2015, QNX Software Systems Limited

mmmd_mdps_list()

Get a list of all loaded MDPs

Synopsis:

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

ssize_t mmmd_mdps_list( char *buffer, size_t buf_len )
```

Arguments:

buffer

A pointer to a string for storing the list of MDP names (may be NULL).

buf_len

The buffer length (may be 0).

Library:

libmd

Description:

This function gets a list of the MDPs successfully loaded and initialized. Calling this function helps diagnose problems with library initialization.

To obtain the buffer length needed to store the list of all loaded MDPs, call this function with *buffer* set to NULL. Use the return value of this first function call to allocate sufficient buffer memory, then call this function a second time, passing in the updated *buffer* pointer to fill in the list of loaded MDPs.

Returns:

>=0

When successful, the function returns either the buffer length needed for storing the MDPs list or the amount of data (in bytes) written to the buffer.

-1

An error occurred (call *mmmd_error_info()* (p. 25) for details).

mmmd_session_close()

Close a metadata-extraction session

Synopsis:

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

int mmmd_session_close( mmmd_hdl_t *hdl )
```

Arguments:

hdl

The handle of the session to close.

Library:

libmd

Returns:

0

Success.

-1

An error occurred (call *mmmd_error_info()* (p. 25) for details).

Copyright © 2015, QNX Software Systems Limited

mmmd_session_open()

Open a metadata-extraction session

Synopsis:

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

mmmd_hdl_t* mmmd_session_open( const char *mediastore,
                               uint32_t flags )
```

Arguments:

mediastore

The URL or mountpoint of the mediastore to associate with the session. The syntax of this argument depends on the mediastore type. For example, to read metadata from a USB stick, set this parameter to **/fs/usb0/** (or something similar). To read metadata from files stored in the root directory of your local filesystem, set this parameter to **/**.

flags

Must be 0; reserved for future use.

Library:

libmd

Description:

This function opens a metadata-extraction session with **libmd**. The session is associated with the media device named in *mediastore*, meaning that you can use it to read metadata from items stored on that device.

Returns:

A valid, non-null session handle

Success.

NULL

Failure (*errno* is set).

mmmd_session_params_set()

Set parameters for a metadata-extraction session

Synopsis:

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

int mmmd_session_params_set( mmmd_hdl_t *hdl,
                             const strm_dict_t *dict )
```

Arguments:

hdl

The handle of the session whose parameters are being set.

dict

A dictionary of key-value pairs representing the parameters. For information on creating dictionaries and storing key-value pairs, see the Dictionary Object API section in the *Multimedia Renderer Developer's Guide*.

Library:

libmd

Description:

This function sets parameters for a metadata-extraction session. After these parameters are set, they can't be unset or changed. Also, they apply only to MDPs that haven't been already used in the current session, so you should call this function just after calling *mmmd_session_open()* (p. 37) but before calling *mmmd_get()* (p. 30).

Currently, only the MMF MDP uses session parameters, which it passes to the Addon Interfaces Library (**libaoi**) when configuring streamers for reading files from HTTP servers. When **libmd** uses other MDPs to read metadata, session parameters defined through this API call have no effect. You should therefore set session parameters only if you want to read metadata from HTTP servers.

The session parameters that you can apply to MMF are the same as the HTTP-related options that you can define as context, input, or track parameters in the Multimedia Renderer API.

Returns:

0

Success.

-1

An error occurred (call *mmmd_error_info()* (p. 25) for details).

Copyright © 2015, QNX Software Systems Limited

mmmd_terminate()

Terminate the library

Synopsis:

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

int mmmd_terminate( void )
```

Library:

libmd

Description:

This function terminates the library from use by unloading all MDPs. You must call this function once and it must be the last function you call.

Returns:

0

Success.

-1

An error occurred (*errno* is set).

Index

Copyright © 2015, QNX Software Systems Limited