Multimedia Synchronizer 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 20, 2015

Contents

About This Guide

The *Multimedia Synchronizer Developer's Guide* is intended for developers who want to write multimedia applications that use the **mm-sync** service to synchronize a device's media information with a database.

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

To find out about:	Go to:
The multistep process used for synchronizing mediastores	*The synchronization process* (p. 10)
Running the **mm-sync** service	*Setting up the Multimedia Synchronizer Environment* (p. 19)
The command-line utility for managing synchronizations	*mmsyncclient command utility* (p. 23)
The application steps needed to synchronize a mediastore's metadata to a dynamically loaded database	*Synchronizing media content from applications* (p. 32)
Controlling and troubleshooting synchronizations	*Working with Synchronizations* (p. 31)
Defining what content gets synchronized and how media information gets stored in databases	*Configuring Mediastore Synchronization* (p. 41)
The API data structures and functions to use for managing synchronizations and for interpreting **mm-sync** errors	*Multimedia Synchronizer API* (p. 63)

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 Synchronization Overview

The multimedia synchronizer service, **mm-sync**, synchronizes mediastore information with the contents of persistent storage. Synchronization involves extracting file and media metadata from devices and uploading that information into QDB databases.

Each mediastore has its own database that contains metadata such as the media types of its individual files, track names, image dimensions, and so on. Using databases for storage offers applications an efficient way of obtaining media metadata because they can simply query databases instead of searching the filesystems of external devices and then retrieving file information through system calls.

Another advantage to this design is that applications can view media information about any device that has been synchronized, even if that device is no longer present on the system.

The **mm-sync** service can:

- index files
- index metadata
- resolve playlists
- process or index external databases

Client applications can use **mm-sync** to synchronize some or all mediastore content at any time through API calls. The extracted media information helps these applications to find, organize, and play media, as well as provide end users with helpful and up-to-date information.

The **mm-sync** service selects the best synchronizer for a given mediastore to ensure users have the most accurate and complete metadata. The synchronization proceeds in steps and uses the QDB database server to transfer the uploaded information into persistent storage.

Artwork extraction

Although the media content that **mm-sync** synchronizes to databases may include artwork images, the synchronizer service does *not* extract artwork. To do this, your client must use **libmd** to retrieve the artwork from the media files, based on the list available in the mediastore database previously synchronized by **mm-sync**. Information on **libmd** artwork extraction is found in the "Extracting artwork" section of the *Metadata Provider Library Reference*.

Mediastore detection

Mediastore insertions or removals are *not* detected by **mm-sync**. Applications can make system calls (e.g., *readdir()*) to search for attached mediastores that are mounted in the local filesystem, and then manually synchronize their contents with **mm-sync**.

Changes in device filesystems, such as file additions or removals, are also not detected by **mm-sync**. Applications must explicitly synchronize mediastore content to guarantee up-to-date file information for their users. An advantage of this approach is that **mm-sync** won't do a synchronization for every mediastore insertion or change of content, which would consume many system resources and likely slow down client applications.

The synchronization process

The multimedia synchronization is done in multiple passes, so that the application may begin playing media without having to wait for all the metadata to be synchronized. You can perform various types of synchronizations that update some or all of the media information, depending on your application needs.

Before the synchronization begins, **mm-sync** first selects the best synchronizer for the requested operation. The service then traverses the files and folders on the mediastore in multiple passes to extract and upload the file information and media metadata. The number of passes made depends on the synchronization parameters. Each pass reads different media information and populates the appropriate database tables.

Your client application can do other work while the synchronization completes. The **mm-sync** API functions are nonblocking so the same thread that starts a synchronization can go on to perform other tasks, including playing media once the file information has been uploaded. Also, if the synchronizer service doesn't have the resources to start a synchronization, it places the request in its "pending" queue and signals the client, which can monitor the synchronization progress through event notifications.

Synchronizer selection

The **mm-sync** service provides many synchronizers designed for various media and storage devices. Some synchronizers can extract the metadata from a certain device, media type, or playlist better than other synchronizers. When **mm-sync** receives a synchronization request, it selects the best synchronizer to use for the content being synchronized and for the device and media type.

For example, for a CDDA:

1. The **mm-sync** service checks if the CD device supports CD-Text and if the Gracenote plugin is enabled.
2. If CD-Text or Gracenote support is available, **mm-sync** uses the synchronizer most appropriate for extracting the metadata of the CD files.
3. If no such synchronizer is available, **mm-sync** uses its default synchronizer to get the metadata.

Once **mm-sync** has selected the synchronizer, the service begins updating media information in the database in a multipass process.

Synchronizer ratings

Synchronizers rate themselves based on their ability to handle different types of mediastores, files, and playlists. When it receives a request for a synchronization, **mm-sync** queries synchronizers for their ratings and compares them to identify the best synchronizer for the task.

Internally, ratings are expressed as integer values of 0 or greater. A value of 0 means the synchronizer doesn't support the device. A value of 1 means it supports the device but is the worst choice, while higher values mean the synchronizer is a better choice. There's no official highest rating, but typically values are between 0 and 100.

The rating values used by synchronizers are similar to those used by metadata plugins, which are libraries that define their own synchronizers that retrieve metadata, sometimes from third-party sources such as MusicBrainz. You can override the default ratings of metadata plugin synchronizers in the configuration file, but you can't override the ratings of the built-in **mm-sync** synchronizers.

Manual selection of a synchronizer

You can instruct **mm-sync** to use a specific synchronizer to bypass the automated selection based on ratings.

To request to use a synchronizer from a client application, set the *use_synchronizer* or *force_synchronizer* option in the *mm_sync_start()* API call or `sync_start` command.

The built-in supported synchronizers are:

- **audiocd**
- **bfsrecurse**
- **custom**
- **dbm**
- **devb**
- **dvdaudio**
- **dvdvideo**
- **internet**
- **mediafs**
- **mediafs2wire**
- **vcd**

Related Links

Synchronization passes (p. 11)
The **mm-sync** service synchronizes media information in three passes known as the files, metadata, and playlist passes. The default behavior is to perform all three passes but you can perform any subset of these passes by setting the right flags when requesting a synchronization.

mmsyncclient command utility (p. 23)
Manage mediastore synchronizations

mm_sync_start() (p. 75)
Start a synchronization

Synchronization passes

The **mm-sync** service synchronizes media information in three passes known as the files, metadata, and playlist passes. The default behavior is to perform all three passes but you can perform any subset of these passes by setting the right flags when requesting a synchronization.

Each pass uploads different information into the database. For the metadata pass, the tables updated depend on the media types of the files being synchronized. The following table shows the database tables updated during the three synchronization passes, based on the default database schema:

Pass	Media type	Tables updated
Files	All	**files**, **folders**, **playlists**, **mediastore_metadata**
Metadata	Audio	**audio_metadata**, **genres**, **artists**, **albums**, **mediastore_metadata**

Pass	Media type	Tables updated
	Video	**video_metadata**, **genres**, **artists**, **mediastore_metadata**
	Photo	**photo_metadata**, **mediastore_metadata**
Playlist	All	**files**, **folders**, **playlists**, **playlist_entries**, **mediastore_metadata**

Files pass

The files pass retrieves basic file information and then updates the **files**, **folders**, and **playlists** table entries for all media files and playlists found on the device. In this pass, **mm-sync** removes from the database any files, folders, or playlists that used to exist but are no longer on the device. For deleted playlists, **mm-sync** also removes their playlist entries.

No metadata has been gathered at this point, but the file information obtained lets you begin playing media.

At the end of each pass, **mm-sync** updates the *syncflags* and *last_sync* fields in the **mediastore_metadata** table to mark synchronization progress.

Metadata pass

The metadata pass retrieves metadata associated with the files on the device. Metadata can include running time, display details, author, and other creation and playback information. The table entries that get updated depend on the media types (audio, video, or photo) of the mediastore files whose metadata is synchronized.

In the database schema definition file, you can change the tables that hold the metadata for different file types. The default configuration splits metadata for audio, video, and photo files into different tables, but you could define more or fewer tables to store the metadata, depending on the file types and media information you want to support. The **files** table is always present, but you can add fields to store more information in this table.

After the metadata pass, metadata is accurate for the mediastore files being synchronized, and you can display this information to the user.

Playlist pass

The playlist pass converts playlist entries into ordered lists of file IDs, which are stored in the **playlist_entries** table.

In this pass, **mm-sync** tries to match the filename in each playlist entry with a filename in the database. For playlists on devices with media-based filesystems, **mm-sync** searches for up to 100 database matches of a playlist filename (any matches beyond 100 matches are ignored), and then associates the playlist entry with the database file that is the best match.

For new playlists, you must synchronize the playlist file by running the files pass with the appropriate synchronization path before running the playlist pass. This is because the new playlist must have an entry in the **playlists** table for the playlist pass to be able to resolve the file IDs of the playlist's entries.

If a playlist entry refers to a file no longer on the device, **mm-sync** doesn't delete the entry but instead sets its file ID (*fid*) to 0. To delete playlist entries that refer to nonexistent files, you must remove the names of those files from the playlists on the device and then run the playlist pass again.

When this pass has completed, you can display and use playlists.

Related Links

Synchronizer selection (p. 10)
The **mm-sync** service provides many synchronizers designed for various media and storage devices. Some synchronizers can extract the metadata from a certain device, media type, or playlist better than other synchronizers. When **mm-sync** receives a synchronization request, it selects the best synchronizer to use for the content being synchronized and for the device and media type.

Mediastore filesystem traversal (p. 13)
At each synchronization pass, **mm-sync** traverses the mediastore filesystem to extract and upload media information into the mediastore's database. The section of the filesystem tree that is synchronized depends on the user-specified path. If a blank path is given and the recursive option is set, the entire filesystem is synchronized. Otherwise, only the files and folders named by the path are synchronized.

Database cleanup (p. 14)
During synchronization, **mm-sync** may clean up the database to remove references to files no longer on the mediastore and unused references to the metadata of these files. The cleanup ensures the accuracy of the content and responsiveness of the database by eliminating unneeded, stale data.

Optimization of synchronization for slow devices (p. 15)
Some devices can store large volumes of information that isn't all media content or that may be inherently slow to read. To avoid an unacceptably long delay between when the device is inserted and when media playback can begin, **mm-sync** optimizes the synchronization of slow devices through its foreground merge feature.

Full, directed, and file synchronizations (p. 17)
The multimedia synchronizer doesn't provide separate controls for synchronizing an entire mediastore versus certain folders or files. You use the same function call to synchronize content whether it's a full, recursive synchronization of all the mediastore content or of only a folder, file, or playlist.

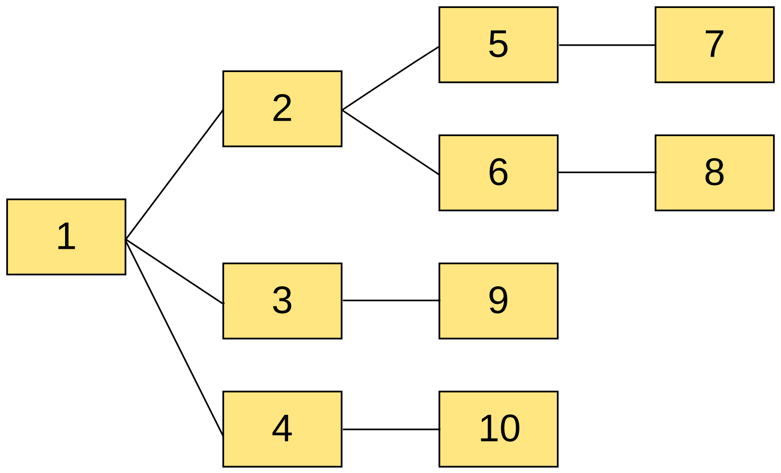
Mediastore filesystem traversal

At each synchronization pass, **mm-sync** traverses the mediastore filesystem to extract and upload media information into the mediastore's database. The section of the filesystem tree that is synchronized depends on the user-specified path. If a blank path is given and the recursive option is set, the entire filesystem is synchronized. Otherwise, only the files and folders named by the path are synchronized.

The filesystem traversal is done with a breadth-first walk of the mediastore's directory structure, as follows:

1. Start at the root node of the section of the filesystem tree being synchronized. When a blank path is given, the root is the mediastore's root folder. Synchronize this root object before synchronizing any others.
2. Root objects that are folders can contain other objects, which may be files or subfolders. Synchronize any objects in the root before examining the contents of any of those objects.
3. If the recursive option is set, for every contained object that is a folder, traverse its directory structure to synchronize its contents before starting on the next folder.

The following image shows the synchronization order for typical contents of a mediastore filesystem:



You can enable or disable the recursive option (MMSYNC_OPTION_RECURSIVE) in the call to *mm_sync_start()* or in the `sync_start` command. By default, this option is disabled, so you must enable it when you want to synchronize the entire mediastore or a whole directory subtree.

This traversal policy ensures that all folders with the same parent node are synchronized before folders deeper in the tree are examined. So **mm-sync** will display information on the contents in the root folder before it retrieves information on files in subfolders. This progressive information retrieval makes sense because end users often start at the filesystem root when they first access mediastores, and then descend into subfolders as they explore the media content.

If you cancel a synchronization in progress, some folders may be fully synchronized while others may not have any of their contents synchronized.

The synchronization path provided by the client to **mm-sync** determines the synchronization startpoint (the root node in the walk). The path is relative to the mediastore filesystem. For example, the path *The_Doors* tells **mm-sync** to locate the object named *The_Doors* within the root folder. If the name refers to a file, only that file gets synchronized. If the name refers to a folder, the files it contains get synchronized; when the recursive option has been set, the subfolders also get synchronized.

Database cleanup

During synchronization, **mm-sync** may clean up the database to remove references to files no longer on the mediastore and unused references to the metadata of these files. The cleanup ensures the accuracy of the content and responsiveness of the database by eliminating unneeded, stale data.

The multimedia synchronizer may attempt to clean up unused data if the database is "prunable" and the current synchronization is not the first synchronization of the mediastore.

A database is prunable if its configuration allows for unused data to be deleted. Pruning is the name of the technique used during database cleanup to incrementally remove unneeded database entries.

The files pass (the first synchronization pass) identifies the media currently stored on the device. As part of this activity, **mm-sync** deletes from the **files**, **folders**, **playlist**, and **playlist_entries** tables all entries for content not found on the mediastore. Any metadata for this nonexistent content is deleted from the **audio_metadata**, **video_metadata**, and **photo_metadata** tables or from any metadata tables defined in a custom configuration. The database then remains a manageable size and its content reflects what is stored on the media.

The playlist pass (the third synchronization pass) resolves the playlist information. During this pass, **mm-sync** deletes from the **playlist** and **playlist_entries** tables all entries for playlist content no longer on the mediastore.

The cleanup continues as **mm-sync** then prunes the **audio_metadata**, **video_metadata**, **photo_metadata**, **genres**, **artists**, and **albums** tables to delete the metadata for files removed from the media.

The cleanup can take up to several seconds, depending on the size of the database for the device being synchronized. Clients monitoring the synchronization might therefore see a delay between the receipt of the event signaling the completion of the playlist pass (the third pass) and the event signaling the completion of the entire synchronization operation. Furthermore, the QDB database service, which manages the databases **mm-sync** uses, can consume a large portion of CPU resources throughout the operation.

Related Links

Synchronization passes (p. 11)
The **mm-sync** service synchronizes media information in three passes known as the files, metadata, and playlist passes. The default behavior is to perform all three passes but you can perform any subset of these passes by setting the right flags when requesting a synchronization.

Optimization of synchronization for slow devices (p. 15)
Some devices can store large volumes of information that isn't all media content or that may be inherently slow to read. To avoid an unacceptably long delay between when the device is inserted and when media playback can begin, **mm-sync** optimizes the synchronization of slow devices through its foreground merge feature.

Optimization of synchronization for slow devices

Some devices can store large volumes of information that isn't all media content or that may be inherently slow to read. To avoid an unacceptably long delay between when the device is inserted and when media playback can begin, **mm-sync** optimizes the synchronization of slow devices through its foreground merge feature.

The overhead of database transactions makes it valuable to insert information for many files (say, 100) in one database operation. If the media file information on the device is slow to read, extracting information for the number of files necessary for a full transaction can take up to several seconds, easily exceeding the client application's limit on playback response times. To allow faster playback, **mm-sync** uploads information on the first media file found, which gets marked as the *first fid* file (first playable file), in a dedicated database transaction. This operation is known as a foreground synchronization merge.

The merge makes the first media file playable sooner and allows the remaining file and metadata information to be synchronized later in the background, while the first media file is played. When developing your client application, you can choose either to begin playback when the *first fid* is synchronized to the database or to wait until the files pass of synchronization completes so you have the entire list of tracks. The second strategy may be necessary if you have to play tracks in a certain order (say, alphabetically) that requires that all files are synchronized before playback can start.

You can set the priority of the synchronization merge with the **<MergePriorityAdjust>** element in the configuration file.

Related Links

Synchronization passes (p. 11)
The **mm-sync** service synchronizes media information in three passes known as the files, metadata, and playlist passes. The default behavior is to perform all three passes but you can perform any subset of these passes by setting the right flags when requesting a synchronization.

Database cleanup (p. 14)
During synchronization, **mm-sync** may clean up the database to remove references to files no longer on the mediastore and unused references to the metadata of these files. The cleanup ensures the accuracy of the content and responsiveness of the database by eliminating unneeded, stale data.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Full, directed, and file synchronizations

The multimedia synchronizer doesn't provide separate controls for synchronizing an entire mediastore versus certain folders or files. You use the same function call to synchronize content whether it's a full, recursive synchronization of all the mediastore content or of only a folder, file, or playlist.

The `sync_start` command for the **mmsyncclient** utility and the *mm_sync_start()* API function both take an argument for specifying the synchronization path. When the path lists a specific file or folder instead of the root folder (**/**), we refer to that synchronization as a *directed synchronization*.

The following table shows the path argument syntax and the additional options needed to perform synchronizations of varying scope:

Scope	Path syntax	Additional options
Entire mediastore	`"/"`	MMSYNC_OPTION_RECURSIVE *(required)*
Existing folder	`"/$folderpath/"` (you can find a folder's path by looking up its *folderid* in the **folders** table, and then reading the *basepath* field from the result row)	MMSYNC_OPTION_RECURSIVE *(optional; use to synchronize contents of all subfolders)*
Existing file	`"/$filepath"` (you can find an existing file's path by looking up its *fid* in the **files** table, reading the result row's *folderid* field, and then looking up the folder's path, as when synchronizing a folder)	MMSYNC_OPTION_PASS_FILES *(required)*, MMSYNC_OPTION_PASS_METADATA *(required)*
New files (including playlists and subfolders)	`"/$folderpath/"` (you can find the path of the folder containing the new media files, playlists, and possibly subfolders by looking up the folder's *folderid* value, as when synchronizing a folder)	MMSYNC_OPTION_PASS_FILES *(required)*, MMSYNC_OPTION_PASS_METADATA *(required)*, MMSYNC_OPTION_RECURSIVE *(optional; use to synchronize contents of all subfolders)*
Entries for an existing playlist	`"/$playlistpath"` (you can find a playlist's path by joining the **playlists** and **folders** tables on the *folderid* field, then reading the *basepath* field from the join result)	MMSYNC_OPTION_PASS_PLAYLISTS *(required)*

Mediastores with a single directory level, such as music CDs, don't support synchronizations directed at folders or files. For these mediastore types, synchronizations must be done for the entire device by setting the path to `"/"`. Only mediastores with hierarchical directory structures, such as HDDs, iPods, USB sticks, and data CDs, support directed synchronizations. For some of these latter mediastore types, the performance may not be as good when synchronizing the entire mediastore as opposed to targeted folders or files.

When to use directed synchronizations

Directed synchronizations are useful when you need information on certain folders or files but you don't want to do an expensive synchronization of all the mediastore content. By directing the synchronization at the folders and files within a path on the mediastore, you can synchronize some media information to make it available sooner to users. You can then synchronize the rest of the information later, if needed.

When a directed synchronization notices a folder that's in the **mm-sync** database but is not on the mediastore, the synchronization deletes the folder and its contents from the database. With this behavior, a client application can remove a folder from a mediastore and then use directed synchronization to remove this folder from the database.

Synchronization of individual files is typically done when an application knows that a specific file change has occurred: a file has been deleted, added, moved, or renamed. Running the files and metadata synchronization passes on that file will update its information in the database.

Cancelling a synchronization in progress

To improve the end user's ability to browse a mediastore, such as an iPod, the **mm-sync** service offers the MMSYNC_OPTION_CANCEL_CURRENT flag. If **mm-sync** is performing a synchronization but your client application needs to respond to a user action, such as navigating to another folder in the mediastore file explorer, your application can set this flag when calling *mm_sync_start()* to cancel the current synchronization and start a new synchronization. The cancellation feature helps reduce resource consumption by allowing you to stop synchronizations that become unnecessary when the application goals change.

Related Links

The synchronization process (p. 10)
The multimedia synchronization is done in multiple passes, so that the application may begin playing media without having to wait for all the metadata to be synchronized. You can perform various types of synchronizations that update some or all of the media information, depending on your application needs.

mmsyncclient command utility (p. 23)
Manage mediastore synchronizations

mm_sync_start() (p. 75)
Start a synchronization

Chapter 2
Setting up the Multimedia Synchronizer Environment

Multimedia synchronization is done by the **mm-sync** utility. Before using **mm-sync**, you must first start both the Persistent Publish/Subscribe (PPS) service and the QNX database (QDB) server, and then load the databases of the mediastores you plan to synchronize. You can then launch **mm-sync** and start synchronizing media content.

To set up **mm-sync**:

1. In a QNX Neutrino terminal, enter `io-fs-media` to start the IO service for reading and writing to RAM-based locations.

   Although it's not required, we recommend running your QDB databases in RAM; for example, from a **tmpfs** filesystem. You can also run databases from locations in QNX filesystems and flash filesystems but performance may suffer with these two filesystem types due to the inherent slowness in writing to the storage media.

2. 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 describe the configuration of the QDB databases.

3. Enter `mkdir -p /pps/qnx/qdb` to create the directory structure used in the PPS configuration path.

4. Enter `qdb` followed by any desired options to start the QDB server.

   For debugging purposes, you should start **qdb** with `-vvvvvvvV` options to get verbose output. The `-v` option is cumulative, with each additional `v` adding a level of verbosity, up to seven levels. The `-V` option sends output to the console and to the **sloginfo** log file.

5. Load the databases that hold the file information and metadata for any mediastores you plan to use. For each database you want to load, you must:

   - Copy an existing configuration object or, from a client application using the *open()* and *write()* system calls, output the list of configuration attributes and values into the **config** subdirectory under the PPS configuration path (**/pps/qnx/qdb/**).

     QDB parses the configuration object's contents and tries to load the database with the same name as the object. QDB then creates a status object that indicates the database state after the loading attempt. If the storage file named in the configuration object doesn't exist, QDB creates the storage file and if directed, populates the database with initial data.

     The directory that will contain the new storage file must exist before you start loading the database. Otherwise, the loading fails and QDB sets the status to **Error**.

   See the *QNX QDB Developer's Guide* for details on database configuration files.

   The **mm-sync** service can now access any of the loaded databases to perform synchronizations on the associated mediastores.

6. Start the multimedia synchronization engine by entering into a terminal a command in the form: `mm-sync -c /patches/640-0315/target/qnx6/etc/mm/mm-sync.conf -vvvv [remaining options]`.

   You can point **mm-sync** to a configuration file by using the `-c` option. Although the **mm-sync** package includes a default configuration file, it's often beneficial (or even necessary) to use a custom configuration based on your system requirements.

The multimedia synchronizer service is running. You can now manage synchronizations by issuing commands to the **mmsyncclient** utility or by launching media applications that detect the insertion of new devices and call the appropriate **mm-sync** API functions.

mm-sync command line

Start multimedia synchronization engine

Synopsis:

```
mm-sync [-c config_file] [-D] [-F] [-m context_path]
    [-o option[,option2...]] [-s] [-S] [-v[v...]] [-V]
```

Options:

-c ***config_file***

Specify an overridden configuration file. The full path of the default configuration file that **mm-sync** looks for is **/etc/mm/mm-sync.conf**, but you can provide a path to any other valid configuration file.

-D

Turn on the debugging output.

-F

Keep the synchronizer process in the foreground. This consumes more CPU resources but is handy when you need to minimize the time for making the media content playable.

-m ***context_path***

Specify an overridden control context path. This is the path of the device object used in synchronizations. This option allows you to run multiple **mm-sync** instances concurrently with the same configuration.

When this option isn't used, **mm-sync** uses the context path provided in the configuration file. If no such path is given in the configuration file or if no configuration file is provided, **mm-sync** uses the system default of **/dev/mmsync**.

-o ***option***

Configure miscellaneous options. Currently, only one option is supported:

sync_verbosity=<0..7>

Set the verbosity level when logging synchronization details. If this option isn't specified, the default behavior is to match the `-v` setting.

-S

Enable logging of synchronization statistics. If you omit this option, the default behavior is to not log these statistics.

-s

Print logs to *stderr* in addition to **sloginfo**.

-V

Print expected schema versions to *stderr* and then quit.

-v

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

The `-v` option is handy when you're trying to understand the operation of **mm-sync**, 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-sync** command line runs the multimedia synchronizer engine with various user-specified parameters. Through parameters, you can specify a control context path and configuration file to use, make the service run in the foreground, and configure many aspects of logging, including the verbosity level and the logging of statistics.

The **mm-sync** service runs as a server process and responds to synchronization commands issued from the **mmsyncclient** utility or API calls made from client applications.

Related Links

mmsyncclient command utility (p. 23)
Manage mediastore synchronizations

Synchronizing media content from applications (p. 32)
Media applications can manually detect mediastore attachments and start synchronizations. To do this, applications must read a PPS object to learn the mediastore mountpoints and, from these mountpoints, explore device filesystems to determine the paths of the files to be synchronized. The mountpoints and synchronization paths are needed by **mm-sync**.

mmsyncclient command utility

Manage mediastore synchronizations

Synopsis:

```
mmsyncclient [-e] <mmsync_dev> <command> [<command args>]
```

Options:

-e

Enable event listen mode. This option causes **mmsyncclient** to print synchronization events to the standard output. You can redirect the output to a file if you want to save the output.

The following is an example of typical output produced when the `-e` option is enabled:

```
Event listen mode, Ctrl + C to exit.
MMSYNC_EVENT_MS_SYNC_STARTED(operation ID 3)
MMSYNC_EVENT_MS_SYNC_FIRST_EXISTING_FID(1, operation ID 3)
MMSYNC_EVENT_MS_1PASSCOMPLETE(operation ID 3)
MMSYNC_EVENT_MS_2PASSCOMPLETE(operation ID 3)
MMSYNC_EVENT_MS_3PASSCOMPLETE(operation ID 3)
MMSYNC_EVENT_MS_SYNCCOMPLETE(operation ID 3)
```

Arguments:

<mmsync_dev>

The device object to use in synchronizations. You must specify the same device used by the **mm-sync** service you started earlier. The default device is **/dev/mm-sync**, but if you overrode this when starting **mm-sync**, the device specified here must match.

<command> [<command args>]

The following synchronization commands are supported:

- *sync_start* (p. 24)
- *sync_suspend* (p. 25)
- *sync_resume* (p. 25)
- *sync_cancel* (p. 26)
- *sync_status_get_byid* (p. 26)
- *sync_status_get_bydbname* (p. 26)
- *sync_status_get* (p. 27)
- *sync_status_get_dbname* (p. 27)
- *sync_debug_set* (p. 27)
- *sync_control* (p. 27)

Description:

The **mmsyncclient** utility allows you to perform synchronizations and monitor their progress through the command line. When you start a synchronization, **mmsyncclient** displays an operation ID (*op_id*), which you can use to pause, resume, or cancel the synchronization or to check its progress at a later time. With the `sync_debug_set` command, you can control how much activity and debugging information is logged, which is useful for troubleshooting.

Commands:

`sync_start`

Start a synchronization of the media content within the specified path on the mediastore that is accessible at the specified mountpoint. The media content is synchronized to the database identified by the device path.

```
mmsyncclient [-e] <mmsync_dev>
        sync_start <db> <mountpoint> <syncpath> <options>
                [<extended_options>]
```

This command accepts the following parameters:

<db>

The device path of the database where the content is to be synchronized.

<mountpoint>

The mountpoint of the mediastore to synchronize.

<syncpath>

The relative path of the file or folder to synchronize.

<options>

The synchronization options (MMSYNC_OPTION_*). You may use one or many of these options to control which synchronization passes get done, whether subfolders are recursively synchronized, and when folder information is updated to reflect the current mediastore files and playlists.

<extended_options>

(Optional) A set of key/value pairs with extended synchronization options, formatted as a comma-separated list of pairs:

```
key1=value1,key2=value2,key3=value3,...
```

Key	Value	Description
`use_synchronizer`	“ *mss name*” (a supported synchronizer, as a string in quotes; e.g., "`dvdaudio`")	Use the requested synchronizer unless it doesn't support the current operation.

Key	Value	Description
`force_synchronizer`	"*mss name*" (a supported synchronizer, as a string in quotes; e.g., `"dvdvideo"`)	Force the use of the requested synchronizer, whether or not it supports the current operation.
`dynamic_folder`	enable | disable	Enable or disable the dynamic setting for the folder listed in *syncpath*. The *fids* for files in this folder will remain constant while this setting is enabled. The setting is nonrecursive, so only the files in the top-level folder in *syncpath* are affected; files in subfolders aren't affected. For information on how this setting impacts synchronization, see "*Maintaining constant IDs for updated files and playlists* (p. 61)".
`metadata_keys`	The metadata fields to be read, as name-value pairs separated by semi-colons: `md_title_name=Poltergeist;md_title_genre=Horror;md_title_album=UnleashTheDemons;md_title_artist=Mr_X`	Retrieve only the listed metadata fields. This setting affects only directed synchronizations in which `dynamic_folder` is enabled. When you define `metadata_keys`, the **libmd** library isn't used for metadata extraction; instead, **mm-sync** sets the metadata fields to the values listed in this option. Note that you must provide values for each field that you list.

sync_suspend

Suspend a synchronization.

```
mmsyncclient [-e] <mmsync_dev> sync_suspend <op_id> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the synchronization to suspend

flags

(Optional) Use MM_SYNC_SUSPEND_FLAGS_WAIT to block until the synchronization thread has been suspended; otherwise, set to 0

sync_resume

Resume a suspended synchronization.

```
mmsyncclient [-e] <mmsync_dev> sync_resume <op_id> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the suspended synchronization to resume

flags

(Optional) Must be 0; reserved for future use

sync_cancel

Cancel a synchronization.

```
mmsyncclient [-e] <mmsync_dev> sync_cancel <op_id> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the synchronization to cancel

flags

(Optional) Must be 0; reserved for future use

sync_status_get_byid

Get the status of a synchronization, based on the operation ID.

```
mmsyncclient [-e] <mmsync_dev> sync_status_get_byid <op_id> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the synchronization whose status is to be returned

flags

(Optional) Must be 0; reserved for future use

sync_status_get_bydbname

Get the status of a synchronization, based on the database name.

```
mmsyncclient [-e] <mmsync_dev>
                    sync_status_get_bydbname <db_name> [flags]
```

This command accepts the following parameters:

<db_name>

The name of the database whose status information is to be returned

flags

(Optional) Must be 0; reserved for future use

sync_status_get

Get the status of all current synchronizations.

```
mmsyncclient [-e] <mmsync_dev> sync_status_get
```

This command has no parameters.

sync_status_get_dbname

Get the name of the database being used in a specific synchronization.

```
mmsyncclient [-e] <mmsync_dev>
                    sync_status_get_dbname <op_id> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the synchronization whose database's name is to be returned

flags

(Optional) Must be 0; reserved for future use

sync_debug_set

Set the logging verbosity and debugging levels.

```
mmsyncclient [-e] <mmsync_dev>
                    sync_debug_set <verbose> <debug>
```

This command accepts the following parameters:

<verbose>

The new verbosity setting to use

<debug>

The new debug setting to use

sync_control

Send commands to a synchronization in progress.

```
mmsyncclient [-e] <mmsync_dev>
                    sync_control <op_id> <extended_options> [flags]
```

This command accepts the following parameters:

<op_id>

The operation ID of the synchronization being controlled

<extended_options>

A set of key/value pairs with synchronization control commands, formatted as a comma-separated list of pairs:

```
key1=value1,key2=value2,key3=value3,...
```

Key	Value	Description
`action`	Currently, only one action is supported: `priority_folder_set`	The action to perform on the synchronization.
`folderid`	An integer storing the ID of a mediastore folder.	The folder the action is performed on. Either this field or *folder_path* must be defined for actions such as `priority_folder_set` that affect a particular folder.
`folder_path`	A string storing the path of a mediastore folder. The path is relative to the mediastore's filesystem (e.g., "**/**" refers to the mediastore's root folder).	The folder the action is performed on. Either this field or *folderid* must be defined for actions such as `priority_folder_set` that affect a particular folder.
`dynamic_folder`	enable | disable	Enable or disable the dynamic setting for the folder referred to by either *folderid* or *folder_path*. This option applies to the `priority_folder_set` action. When this setting is enabled, the *fids* for files in this folder will remain constant. The setting is nonrecursive, so the only files affected are those in the top-level folder in the synchronization path named in the operation *<op_id>*; files in subfolders aren't affected. For information on how this setting impacts synchronization, see "*Maintaining constant IDs for updated files and playlists* (p. 61)".

Action	Description
`priority_folder_set`	Initiates a priority folder synchronization. Requires one of the *folderid* and *folder_path* key/value pairs. You can also define the *dynamic_folder* key to enable or disable the dynamic folder setting.

flags

(Optional) Must be 0; reserved for future use

Returns:

With the `sync_start` command, when the synchronization starts successfully, **mmsyncclient** returns a text string with the operation ID, which allows you to monitor and control the synchronization in follow-up commands. When the synchronization does *not* start successfully, a text string with a failure message and a return code of `-1` is returned.

All other commands return at a minimum a text string with the return code (*rc*) and error number (*errno*). A return code of 0 means the operation completed successfully and is accompanied by an error number of 0. A nonzero return code (typically, `-1`) means an error occurred. See *errno* for which error it was and **sloginfo** for details about the error.

The commands that get synchronization status will return information on the progress of one or many synchronizations, when those commands complete successfully. With the `sync_status_get_byid` and `sync_status_get_bydbname` commands, **mmsyncclient** returns a text string with the following information:

- *<op_id>*, the operation ID of the synchronization
- *Completed*, flags indicating the synchronization passes that have been completed
- *Current*, flag indicating which synchronization pass, if any, is in progress
- *Pending*, flags indicating the synchronization passes that have not yet been started

When either of these two commands fails, **mmsyncclient** returns a text string with the operation ID or database name for the synchronization whose status could not be attained, along with an error string and number. If you provide an operation ID or a database name that doesn't refer to any active synchronization, the standard message containing the return code and error number is returned, with both fields set to 0 because no error actually occurred.

The `sync_status_get` command produces similar output except that when successful, the status of not just one but all active synchronizations is displayed, and the return code is the number of synchronizations in progress or pending. If this command fails, the return code and error number are returned, with *rc* being nonzero and *errno* set based on the error that occurred.

With `sync_status_get_dbname`, if the operation ID refers to an active synchronization, **mmsyncclient** returns a text string naming the database being used in the specified synchronization. Otherwise, the standard message containing the return code and error number is returned. If there's no active synchronization with the given operation ID, the return code is 0, because no error actually occurred. If the command fails, the return code is nonzero and *errno* is set based on the error that occurred.

Related Links

Synchronizer selection (p. 10)
The **mm-sync** service provides many synchronizers designed for various media and storage devices. Some synchronizers can extract the metadata from a certain device, media type, or playlist better than other synchronizers. When **mm-sync** receives a synchronization request, it selects the best synchronizer to use for the content being synchronized and for the device and media type.

mm-sync command line (p. 21)
Start multimedia synchronization engine

Chapter 3
Working with Synchronizations

You can design your media applications to invoke **mm-sync** to update the databases at specific times, based on when certain tracks must become playable. In this case, your applications must determine which media files need to be synchronized and then provide the appropriate path to **mm-sync** to start the synchronization.

Your client applications may need to guarantee a certain level of metadata availability and accuracy to end users. You can meet such user experience goals by using the following three advanced features:

- **progress tracking**, which monitors synchronization pass completion
- **priority folder synchronization**, which allows you to synchronize some content before all other content
- **database verification and repair**, which performs an integrity check and, if necessary, fixes inconsistencies of synchronized metadata

Synchronizing media content from applications

Media applications can manually detect mediastore attachments and start synchronizations. To do this, applications must read a PPS object to learn the mediastore mountpoints and, from these mountpoints, explore device filesystems to determine the paths of the files to be synchronized. The mountpoints and synchronization paths are needed by **mm-sync**.

To detect a mediastore and synchronize its media metadata, your application must:

1. **Monitor mediastore attachments**

   Your application must subscribe to the **/pps/qnx/mount/.all** PPS object to monitor mediastore attachments. When the user attaches a mediastore, the appropriate device publisher writes the device's information to this object. For devices connected through USB (e.g., iPods and USB sticks), the **usblauncher** service publishes the device information; for SD cards, the **mmcsdpub** service publishes it.

2. **Load the database for a newly attached mediastore**

   When your application notices a new entry in the **.all** object, it must read that entry's `id` attribute to obtain the mediastore's unique ID. Your application can then check if the database whose name contains this unique ID is loaded and if not, load that database by writing its configuration object into the QDB database configuration directory (**/pps/qnx/qdb/config/**).

   You must load a database before trying to synchronize or play media content on the corresponding device because the **mm-sync** and **mm-renderer** services can't load databases. For more details on loading databases, see "Loading QDB databases".

3. **Determine the media content to synchronize**

   Before it can determine what content needs to be synchronized, your application must read the `mount` attribute in the **.all** object entry to learn the mediastore's mountpoint. From this mountpoint, your application can explore the mediastore's content and read file information.

   The decision to synchronize specific files or folders can depend on many factors, including their last synchronization time (if known), the available system resources, and the mediastore's type. New files and folders must be synchronized if they contain media that the user might play. Folders with fewer files than before may need to be resynchronized to ensure that their database information is up to date.

4. **Connect to mm-sync**

   If no content needs to be synchronized, your application can skip the rest of these steps and start querying the database to retrieve media metadata and use it as desired. Otherwise, your application must connect to **mm-sync** by calling *mm_sync_connect()* (p. 68), so it can then use the service to upload the media metadata.

5. **Register for mm-sync events**

   Your application can request to receive event notifications by calling *mm_sync_events_register()* (p. 97). The function's *event* argument must be set to a **struct sigevent** object initialized with the type of notification to deliver with each event. Event notifications allow you to track synchronization progress and to learn of any errors that occur.

6. **Start synchronizing media content**

   Your application can now start synchronizing media content by calling *mm_sync_start()* (p. 75). In this funcation call, you must provide the path of the device object used by the mediastore's database, the mediastore's mountpoint, and a synchronization path containing the content that you want to synchronize. The device object is stored in **/pps/qdb/** and has the same name as the database.

   The function call returns a synchronization ID, which you must provide in subsequent API calls (e.g., *sync_status_get_byid()*) to refer to the same synchronization operation.

7. **Monitor mm-sync events**

   To monitor **mm-sync** events, your application must make the appropriate OS system call to wait for the notification defined in Step *5* (p. 32). When it recieves that notification, your application must call *mm_sync_events_get()* (p. 96) to retrieve the event information.

   The MMSYNC_EVENT_MS_SYNCCOMPLETE event means that all the media information that you requested has been uploaded to the database, so you can now use that information.

8. **Disconnect from mm-sync**

   If your application needs to synchronize metadata from another path, it can return to Step *6* (p. 33) to start a new synchronization. The decision to synchronize more metadata might depend on user activity. For instance, if you display a cover flow to represent the available albums, the user could select another album and your application would then need to synchronize the metadata found in the mediastore path that stores the album's tracks.

   When it has synchronized all the media metadata it needs, your application can disconnect from **mm-sync** by calling *mm_sync_disconnect()* (p. 73).

The database for the newly attached mediastore contains the media metadata required by your application. You can issue SQL queries against the database to read the file information and the creation and display information for audio tracks, video files, and photos on the mediastore. You must keep the database in memory for as long as your application needs to access metadata related to the media content on the mediastore. This may be long after you've finished synchronizing the metadata to the database. For details on unloading databases, see “Unloading QDB databases”.

Exploring mediastores through directed synchronizations

You can synchronize content incrementally through *directed synchronizations* (p. 17), which entail giving *mm_sync_start()* a synchronization path of one folder. After a synchronization completes, you can examine the **files** and **folders** database tables and pick a specific subfolder to synchronize in the next *mm_sync_start()* call. In this manner, you can explore a mediastore by using database queries and directed synchronizations.

Related Links

Full, directed, and file synchronizations (p. 17)
The multimedia synchronizer doesn't provide separate controls for synchronizing an entire mediastore versus certain folders or files. You use the same function call to synchronize content whether it's a full, recursive synchronization of all the mediastore content or of only a folder, file, or playlist.

mm-sync command line (p. 21)

Start multimedia synchronization engine

mm_sync_connect() (p. 68)

*Connect to **mm-sync** and obtain a handle*

mm_sync_events_get() (p. 96)

*Get the next queued **mm-sync** event*

mm_sync_events_register() (p. 97)

*Register or unregister for **mm-sync** event notifications*

mm_sync_start() (p. 75)

Start a synchronization

Maintaining database persistence

The **mm-sync** service doesn't create, load, or unload databases. When the user attaches a mediastore, the application must ensure that the mediastore's database is loaded before any synchronization can begin. When the application has finished using a mediastore or the user detaches it, the application can unload the database to free memory.

Client applications must ensure database persistence between uses of specific mediastores. For example, suppose the user plugs a USB stick into their system, then removes and reinserts the same device. The application writer must ensure that the database stays in memory or is reloaded when the USB stick is reinserted, so the user can once again access and play the device's media content. Keeping the database for a recently used device in memory can greatly speed up resynchronizations—recently used devices are likely to be reinserted soon and you can save significant time by not having to reload their databases.

Making databases available to applications

When a mediastore is inserted for the first time, your application must create a database configuration file that names the raw storage file (which has the same name as the database itself) and must copy the configuration file into the QDB database configuration directory (**/pps/qnx/qdb/config/**).

When a mediastore is inserted any subsequent time, the application just needs to copy the configuration file into the same directory to reload the database, as explained in "*Setting up the Multimedia Synchronizer Environment* (p. 19)".

We recommend using the mediastore's unique ID (UID) and the device type (both can be read from the **/pps/qnx/mount/.all** PPS object) to form the database name. The device type should be included in the name to avoid a naming conflict that would arise if two or more devices were assigned the same UID by independent system components that each write to the PPS mountpoint information object.

Suppose the user inserts both an iPod and an audio CD, and both devices are assigned a UID of `4f587192-d2fe-4efb-9fec-cd35531cfa45` by the separate components that publish iPod and CD device information in PPS. In this case, the device databases could be named `iPod-4f587192-d2fe-4efb-9fec-cd35531cfa45` and `audioCD-4f587192-d2fe-4efb-9fec-cd35531cfa45` to eliminate ambiguity.

This naming practice makes it easy to check if the relevant database exists or is loaded, because the same value read from the PPS object can be used in the search string when looking up the database file by name.

Determining when to unload databases

When your client application knows that a mediastore database won't be needed for some time (e.g., when the device has been removed and no other media applications are running) or if memory is low, your client should unload the database to free some memory.

For each database you want to unload, you must:

1. Delete the configuration object with the same name as the database to unload, from the **config** subdirectory under the PPS configuration path (**/pps/qnx/qdb/**).

   QDB removes the database from its control. QDB also deletes the status file in the **status** subdirectory because the database is no longer visible.

2. If the database storage file is stored in RAM, move that file to persistent storage.

   You can speed up media applications by storing your QDB databases in RAM, but you must remember to move the storage files for any unused databases out of RAM and into persistent storage to avoid overusing system memory.

   Keeping inactive databases in persistent storage reduces the memory requirements of media applications while retaining important mediastore metadata. You can copy a device's database back into RAM when you need to access the device's media information again.

Related Links

mm-sync command line (p. 21)
Start multimedia synchronization engine

Tracking synchronization progress

You can track synchronization progress in terms of what passes have been completed for a mediastore by reading the **syncflags** database field or by retrieving the synchronization status through the **mm-sync** API.

The **mm-sync** service updates the **syncflags** field in the **mediastore_metadata** table when a synchronization pass successfully completes. The **syncflags** field uses separate bits to indicate which of the three passes are complete, represented as follows:

- The least-significant bit (bit 0) indicates whether the files pass has been completed (001)
- The next-significant bit (bit 1) indicates whether the metadata pass has been completed (010)
- The next-significant bit (bit 2) indicates whether the playlist pass has been completed (100)

For example, a value of 0 in **syncflags** means that no synchronization passes have been completed; a value of 5 (101) means that the file and playlist passes have been completed, but the metadata pass was skipped. A value of 7 (111) indicates that all three synchronization passes have been completed.

These flags are not cleared if the device is made inactive. When a disk is moved out of the active slot while in a multidisk changer, the disk is not made unavailable, only inactive. Therefore, this action doesn't clear any existing synchronization flags in the **syncflags** field.

API functions for retrieving synchronization status

You can call the *mm_sync_status_get()*, *mm_sync_status_get_byid()*, and *mm_sync_status_get_bydbname()* API functions to monitor synchronization progress. The **mm-sync** API stores synchronization progress information in the **mmsync_status_t** data type, with three fields: **passes_done**, **current_pass**, and **passes_to_do**. These fields have the same format as the **syncflags** database field.

Suppose you start a synchronization and request all three passes, and then cancel the operation just as the files pass is successfully completing. If you then get the synchronization status through the API, the **passes_done** value is 1 (001), the **current_pass** value is 0 (000), and the **passes_to_do** value is 6 (110). These values show that only the first pass was done and that no pass is in progress.

Tracking folder synchronizations

The **mm-sync** process also marks synchronization progress at the folder level by updating the **synced** field in the **folders** table entry for a specific folder as soon as its contents have been synchronized. When the recursive option is set, **mm-sync** starts from the folder named in the path and descends into the subfolders, meaning individual folders get synchronized at different times during each requested pass. The **synced** field allows client applications to know when a given folder has been synchronized, which means its database information will not change.

An alternative to polling the **synced** field is for your client application to register for synchronization events and specify, in the configuration file, which optional folder-based synchronization events to deliver. Tracking progress through folder events is helpful when synchronizing a priority folder.

Related Links

Synchronization passes (p. 11)

The **mm-sync** service synchronizes media information in three passes known as the files, metadata, and playlist passes. The default behavior is to perform all three passes but you can perform any subset of these passes by setting the right flags when requesting a synchronization.

Setting a priority folder (p. 38)
Depending on user actions, your client application may need to interrupt a synchronization in progress and begin synchronizing a new folder. The priority folder synchronization feature helps you reduce the time required for making media content from a certain folder available for viewing or playing.

Repairing database inconsistencies (p. 40)
If you find that the database information for a folder doesn't match what's actually on the mediastore at some point after you synchronized the folder, you can repair the inaccurate database content by calling *mm_sync_start()* or running `sync_start` with special parameters.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

mm_sync_status_get() (p. 78)
Get the statuses of active synchronizations

mm_sync_status_get_bydbname() (p. 80)
Get the status of a synchronization based on the database name

mm_sync_status_get_byid() (p. 81)
Get the status of a synchronization based on the operation ID

Setting a priority folder

Depending on user actions, your client application may need to interrupt a synchronization in progress and begin synchronizing a new folder. The priority folder synchronization feature helps you reduce the time required for making media content from a certain folder available for viewing or playing.

Suppose your application must display up-to-date folder information in a mediastore file browser and, during a synchronization, the user selects a different folder to view or play tracks from. In this case, you should do a priority synchronization of the newly selected folder. This action displays the folder's metadata and makes the folder's media files playable ahead of files in other folders.

To start a priority folder synchronization, call *mm_sync_control()* or run the `sync_control` command with the following extended option settings:

- The *action* key is set to “`priority_folder_set`”.
- Either the *folderid* or *folder_path* key is set to refer to the folder you want to synchronize immediately. You can look up the *folderid* in the **folders** table. If you use *folder_path*, you must provide a relative path within the mediastore's filesystem (e.g., “***/***” refers to the mediastore's root directory).

When it receives such a request, **mm-sync** interrupts any synchronization in progress on the same device on which the priority folder is stored and synchronizes the priority folder's contents before resuming the original synchronization.

The priority folder feature has the following behavior:

- Requests to synchronize the current folder are silently ignored.
- Priority synchronizations can't be recursive; only the priority folder (and not its subfolders) is synchronized before the original synchronization resumes.
- Priority synchronizations can be done only on a mediastore currently being synchronized, because the invocation command requires a valid operation ID.
- The synchronization passes done for the priority folder are the same as those requested in the original synchronization.
- New priority synchronization requests preempt any priority synchronization in progress, meaning the second priority folder gets synchronized, then the first folder (whose synchronization was interrupted) gets synchronized, and then the original synchronization resumes.
- Priority synchronization requests are queued, so if a second request is sent before the synchronization for the first request is started, the second priority folder gets synchronized, then the first folder, and then the original synchronization resumes.

Related Links

Tracking synchronization progress (p. 36)
You can track synchronization progress in terms of what passes have been completed for a mediastore by reading the **syncflags** database field or by retrieving the synchronization status through the **mm-sync** API.

Repairing database inconsistencies (p. 40)

If you find that the database information for a folder doesn't match what's actually on the mediastore at some point after you synchronized the folder, you can repair the inaccurate database content by calling *mm_sync_start()* or running `sync_start` with special parameters.

mm_sync_start() (p. 75)
Start a synchronization

mm_sync_control() (p. 69)
Send commands to a synchronization in progress

Repairing database inconsistencies

If you find that the database information for a folder doesn't match what's actually on the mediastore at some point after you synchronized the folder, you can repair the inaccurate database content by calling *mm_sync_start()* or running `sync_start` with special parameters.

The **mmsync** verification and repair feature can repair inconsistencies in the database fields for a folder and for the files and playlists associated with that folder. If you're aware of many file and playlist additions, removals, or renamings in a mediastore folder since that folder was last synchronized, you may want to check that the data is consistent between the various fields and tables that represent the media files stored in that same folder on the device.

Verifying and optionally repairing folder data is often faster than resynchronizing the folder because the former doesn't entail uploading all file, folder, and playlist information for the folder being checked. Instead, the verification step queries the database to determine if the data in the **folders** table entry is consistent with the data in all **files**, **folders**, and **playlist** table entries that refer to that folder. The service logs any inconsistencies found and any information that is unable to be retrieved. The repair step updates database fields to eliminate any data inconsistencies between the related table entries and then logs this activity.

To verify the database information for a folder, look up the folder's path in the **folders** table, then call *mm_sync_start()* or run the `sync_start` command with this retrieved path as the path argument and with the MMSYNC_OPTION_VERIFY flag enabled.

To perform the extra step of repairing inconsistencies in the folder data, enable the MMSYNC_OPTION_REPAIR flag in addition to the verification flag. If **mm-sync** is unable to repair any inconsistencies, it notes the incomplete repair work in the system log. In such cases, there's probably a database problem that requires immediate attention.

You can verify and repair the database information not just for one folder but for a folder and all its subfolders by setting the MMSYNC_OPTION_RECURSIVE flag in the command or API call.

Related Links

Tracking synchronization progress (p. 36)
You can track synchronization progress in terms of what passes have been completed for a mediastore by reading the **syncflags** database field or by retrieving the synchronization status through the **mm-sync** API.

Setting a priority folder (p. 38)
Depending on user actions, your client application may need to interrupt a synchronization in progress and begin synchronizing a new folder. The priority folder synchronization feature helps you reduce the time required for making media content from a certain folder available for viewing or playing.

mm_sync_start() (p. 75)
Start a synchronization

Chapter 4 Configuring Mediastore Synchronization

The **mm-sync** configuration file is an XML file whose elements and attributes control how media content gets synchronized from devices into databases. An XML configuration file is a convenient way of defining and enforcing policies on what gets synchronized and how media information is represented in the database.

A default configuration file (**/etc/mm/mm-sync.conf**) is included in the product. This file includes comments that describe the purposes of its XML elements (tags) and attributes. It also contains default values for the various configuration settings, most of which are expressed as comments. To enable a configuration setting, uncomment its tag.

You can define your own configuration file to customize synchronization behavior to suit your system's needs. In the configuration file, you can:

- set synchronization thread priorities to adjust the speed of synchronizations and what resources they consume
- limit how many files in one folder can be synchronized and how many directory levels can be searched
- filter the synchronization of media files and playlists based on file extensions
- map the fields received from metadata providers to storage fields in the database
- manage the database size by setting a threshold size value for stopping metadata synchronization

The settings in the default **mm-sync** configuration file are presented in table format. Each table row provides the tag name, attribute, default value, and functional description for one configuration setting. When the **Attribute** column is blank, the setting in the **Default** column refers to the element (tag) value. When the **Attribute** and **Default** columns are both blank, the referenced element contains other elements, whose purposes are summarized in the **Description** column.

Configuration file contents

The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

Numeric settings are expressed as element values between the opening and closing tags. Boolean settings, filenames, and database fields are assigned to element attributes. The **mm-sync** service uses system default values for any required settings it can't find in the configuration file or for those that have invalid values.

The configuration file consists of the XML version declaration (e.g., **<?xml version="1.0" ?>**) followed by the **<Configuration>** root element, which contains all other elements.

The <Configuration> element

The **<Configuration>** element defines the device path used by **mm-sync** and contains the **<Database>** element, which controls the synchronization process behavior.

The contents of the **<Configuration>** element are:

Tag name	Attribute	Default	Description
<ControlContext>	*Path*	**/dev/mm-sync**	Sets the path of the device object used in synchronizations. This path can be overridden in the **mm-sync** command line.
<Database>			Defines some general database settings and the settings that control media synchronization and database size.

Related Links

The <Configuration>/<Database> element (p. 42)
The **<Configuration>/<Database>** element defines some general database settings and the settings that control media synchronization and database size.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

The <Configuration>/<Database>/<Prune> element (p. 50)
The **<Configuration>/<Database>/<Prune>** element defines values that influence how **mm-sync** monitors the database size and stops synchronization of metadata if the database gets too big.

The <Configuration>/<Database> element

The **<Configuration>/<Database>** element defines some general database settings and the settings that control media synchronization and database size.

The contents of this element are:

Tag name	Attribute	Default	Description
<TimebaseSet>	*enabled*	false	Sets automatic adjustment of the internal timebase. For systems that don't have a stable realtime clock, enabling this option causes the synchronization thread to adjust its internal timebase so that all time values used in the database remain monotonically increasing.
<Timeout>		0	Sets a timeout for the database (in milliseconds). A setting of 0 disables the timeout; nonzero values enforce it. Note that nonzero values can create errors because of operations that take longer than the specified timeout.
<CharacterEncodingConverter>	*dll*	`custom_char_converter.so`	Overrides the character-encoding conversion with another library that implements similar functionality.
<Synchronization>			Defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.
<Prune>			Defines values that influence how **mm-sync** monitors the database size and stops synchronization of metadata if the database gets too big.

Related Links

The <Configuration> element (p. 42)
The **<Configuration>** element defines the device path used by **mm-sync** and contains the **<Database>** element, which controls the synchronization process behavior.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

The <Configuration>/<Database>/<Prune> element (p. 50)
The **<Configuration>/<Database>/<Prune>** element defines values that influence how **mm-sync** monitors the database size and stops synchronization of metadata if the database gets too big.

The <Configuration>/<Database>/<Synchronization> element

The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

The elements that define the synchronization settings make up the bulk of the configuration file. It's helpful to visualize the synchronization elements based on the area of functionality they control.

Resource optimization elements

Tag name	Attribute	Default	Description
<Priority>		0	Sets the priority of the foreground synchronization threads. A setting of 0 enforces the default priority, which is the same priority that the **mm-sync** service uses at startup. Nonzero values assign absolute priorities.
<MergePriorityAdjust>		1	Sets the synchronization merge thread's priority adjustment. This value is added to the synchronization thread priority to derive the merge thread priority. The default behavior is to add 1 to the synchronization thread priority.
<MaxThreads>		8	Sets the maximum number of foreground synchronization threads permitted to run at a time.
<MaxRecursionDepth>		8	Sets the maximum directory structure depth to recursively visit when synchronizing a mediastore. This setting also applies to priority folders.
<MaxSyncBuffers>		250	Sets the maximum number of synchronization records in the buffers between the foreground and background synchronization threads. More buffers increases memory usage, but speeds up synchronization.

Database size management elements

Tag name	Attribute	Default	Description
<ChangedFilesHaveConstantId>	*enabled*	off	When enabled, ensures media and playlist files with a changed size or modification date keep the same ID value in the database. In such cases, the **files** and **playlists** table entries for the changed items have their *accurate* flags cleared, but there's no other indication that these items have changed. This setting helps limit the database size by preventing new rows from being created whenever a file is changed, which happens when the ID values are *not* kept the same. For more information on how this setting impacts synchronization, see *Maintaining constant IDs for updated files and playlists* (p. 61).
<StopOnDbLimit>	*enabled*	false	Controls whether or not synchronization stops when a database limit is reached.

Event elements

Tag name	Attribute	Default	Description
<Events>	*folder*	trimonly	Controls what optional synchronization events are delivered. Currently, only the delivery of folder events is controllable. Acceptable values are `on`, `off`, and `trimonly`. You should use `trimonly` to send folder events only when a folder object (file, folder, playlist) is deleted. Use `on` to send events following folder object additions, modifications, and deletions. Use `off` to disable event sending.
<MaxFirstFidEvent>		1	Sets the maximum number of MMSYNC_EVENT_MS_SYNC_FIRST_EXISTING_FID events sent during synchronization. These events are sent during the files pass when **mm-sync** finds a media file that's playable and is already in the database. A setting of *n* causes **mm-sync** to return this event for the first *n* existing files found.

Synchronization filter elements

Tag name	Attribute	Default	Description
<SyncFileMask>		(empty)	Defines a POSIX regular expression (regex) pattern for naming files you do *not* want synchronized. Only one **SyncFileMask** can be specified. The default setting is an empty mask, meaning no files will be skipped during synchronization.
<NonMediaItems>			Controls the prescan done on each folder. You can configure **mm-sync** to skip the synchronization of folders found to have too much nonmedia content during the prescan. The prescan parameters are set through the elements contained in the **<NonMediaItems>** element.
<NonMediaItems>/<MaxItems>		0	Limits how many nonmedia files can be found in a folder before that folder is skipped from synchronization. A setting of 0 disables this limit.
	consecutive	false	Sets whether the limit of nonmedia files is a consecutive file limit. The file order is generally not guaranteed, so using a consecutive limit could yield nondeterministic results. When this setting is disabled, as is the default, **MaxItems** is a limit of total files.
<NonMediaItems>/<PrescanLimit>		0	Limits how many files can be scanned when searching for media content in a folder. A setting of 0 disables this limit and forces the entire folder to be scanned; nonzero values enforce a limit on the files scanned.

Tag name	Attribute	Default	Description
<MaxFolderItems>		0	Controls the maximum number of items read from a folder. This limit excludes "." and "..", but includes any items whose filenames match the pattern in **<SyncFileMask>**. A setting of 0 disables this limit; nonzero values enforce it.
<MaxMediaStoreItemsConfiguration>			Controls the maximum number of items read from mediastores. For each mediastore on which you want to impose a limit of items read, define a separate **<MaxMediaStoreItems>** element.
<MaxMediaStoreItemsConfiguration>/ <MaxMediaStoreItems>		100	Limits the number of items read from a mediastore. This limit excludes "." and "..", but includes any items whose filenames match the pattern in **<SyncFileMask>**. A setting of 0 disables this limit; nonzero values enforce it.
	mediastore	(none)	Names the mediastore the limit of items read applies to.
<extensions>			Specifies which file extensions are supported, allowing you to filter the synchronization of content by file type. This element contains the **<playlists>** element, which lists supported playlist extensions, and the **<library>** element, which lists supported media file extensions. Files with unlisted extensions aren't synchronized.

Advanced configuration elements

Some synchronization elements contain many other elements that collectively configure helper utilities such as plugins and playlist synchronizers used by **mm-sync** to perform advanced tasks.

Name	Description
<ConfigurableMetadata>	Defines the metadata provider fields that are copied into specific database fields. The **mm-sync** service has no special knowledge of this metadata but will pass it along to the database.
<MSS>	Contains elements that configure mediastore synchronizers (MSSs). Currently, only the synchronizers for Apple devices can be configured in this area.
<PLSS>	Defines resource usage limits and filename-matching parameters for playlist session synchronizers (PLSSs). This element can also name a nondefault configuration file for the library that **mm-sync** uses to synchronize playlists.

Related Links

The <Configuration> element (p. 42)

The **<Configuration>** element defines the device path used by **mm-sync** and contains the **<Database>** element, which controls the synchronization process behavior.

The <Configuration>/<Database> element (p. 42)
The **<Configuration>/<Database>** element defines some general database settings and the settings that control media synchronization and database size.

The <Configuration>/<Database>/<Synchronization>/<MSS> element (p. 49)
The **<Configuration>/<Database>/<Synchronization>/<MSS>** element configures mediastore synchronizers (MSSs) for specific mediastore types. Currently, only the synchronizers for Apple devices can be configured by this element.

The <Configuration>/<Database>/<Synchronization>/<PLSS> element (p. 49)
The **<Configuration>/<Database>/<Synchronization>/<PLSS>** element defines playlist session synchronizer (PLSS) settings related to resource usage and playlist entry matching. This element can also specify a configuration file for the **mmplaylist** library, which **mm-sync** uses to synchronize playlists.

The <Configuration>/<Database>/<Prune> element (p. 50)
The **<Configuration>/<Database>/<Prune>** element defines values that influence how **mm-sync** monitors the database size and stops synchronization of metadata if the database gets too big.

Skipping files based on their names (p. 53)
The **<SyncFileMask>** element allows you to ignore files during synchronization based on a character string in the filename (including the file extension).

Limiting the number of items read (p. 59)
You can limit the total number of items read from any folder or the entire mediastore. An excessively large number of items can make using the device very slow and can cause poor response times for synchronization operations such as changing priority folders or determining mediastore changes.

Maintaining constant IDs for updated files and playlists (p. 61)
The **<ChangedFilesHaveConstantId>** element in the configuration file and the `dynamic_folder` option in the command for starting a synchronization both force **mm-sync** to keep the same IDs in the database for media and playlist files that have been modified.

Filtering synchronization by file type (p. 55)
Client applications can filter the types of media files and playlists that get synchronized. The **<extensions>** element contains the **<library>** and **<playlists>** elements, which list the extensions that media files and playlists must have to be synchronized. Mediastore files with unlisted extensions don't get synchronized.

The <Configuration>/<Database>/<Synchronization>/<ConfigurableMetadata> element

The **<Configuration>/<Database>/<Synchronization>/<ConfigurableMetadata>** element configures metadata support. For each metadata field you want to store in the database, you must define a **<metadata>** element, contained in **<ConfigurableMetadata>**, that maps the field read from the metadata provider to the database storage field and defines other metadata parameters.

The **<metadata>** elements have the following attributes:

Attribute	Description
ftype	Declares the media type for the metadata. Acceptable values are `audio`, `video`, and `photo`.

Attribute	Description
table	Names the database field that stores the metadata you want to synchronize. The table name is listed first, followed by a dot (.), and then the field name. For example, a *table* value of `video_metadata.width` tells **mm-sync** that the metadata is destined for the *width* field in the **video_metadata** table.
ext_table	*(Optional)* Identifies the table and field (column) that stores the values for this bit of metadata. This table is considered external because it's separate from the main metadata table. For instance, an *ext_table* setting of `artists.artist` instructs **mm-sync** to store the metdata value (in this case, the artist name) in the *artist* column of the **artists** table. When you define this attribute, you must also define *ext_table_rel*.
ext_table_rel	*(Optional)* Identifies the field that relates external table entries to metadata table entries. For instance, an *ext_table_rel* setting of `artists.artist_id` instructs **mm-sync** to store the *artist_id* field (instead of the artist name) in the metadata table and to store the artist name in the external table entry with the matching *artist_id*. When you define this attribute, you must also define *ext_table*.
md_map	Names the metadata provider fields to synchronize to the database. The metadata provider name is listed first, followed by the AT sign (@), and then a comma-separated list of applicable field names. Details for multiple metadata providers must be separated by a semicolon. Consider this example: `libmd@artist,albumartist;ipod@artist;` This setting tells **mm-sync** that the current mapping applies to the *artist* and *albumartist* fields from the **libmd** metadata provider and the *artist* field from the **ipod** metadata provider. Note that metadata is matched in left-to-right order. In this example, then, if both fields are available from **libmd**, only the *artist* field will be used from this provider.
maximum	*(Optional)* Limits the maximum number of bytes for a metadata string that **mm-sync** writes in the field named by *table*. By default, the length limit is 256 characters.

Storing metadata field values in external tables

For metadata fields with values repeated for many media files, such as artist or genre, you can save space and improve performance by storing those field values in an external table and using an index in the main metadata table to refer to specific values.

Suppose a mediastore contains several tracks with an artist setting of “Jamiroquai”. Instead of replicating this string in many metadata table rows, you can store a single copy of the artist name with its associated index in an external table, and then store the index in all metadata table entries representing tracks by “Jamiroquai”. Storing indexes instead of strings reduces the size of metadata table rows and allows you to sort the metadata faster because index comparisons are considerably faster than string comparisons.

For an individual media file, you can find its setting for any metadata field whose values are stored in an external table by joining the metadata table with the external table on the field named by *ext_table_rel*.

Suppose you have an external table **genres** for storing the names of musical genres and a field **genre_id** for relating the external table entries to the **audio_metadata** table entries. If you want to list all tracks for a given genre, say "Pop", you can join these two tables on the **genre_id** field and filter the results based on the genre name "Pop".

The <Configuration>/<Database>/<Synchronization>/<MSS> element

The **<Configuration>/<Database>/<Synchronization>/<MSS>** element configures mediastore synchronizers (MSSs) for specific mediastore types. Currently, only the synchronizers for Apple devices can be configured by this element.

The contents of the **<MSS>** element are:

Tag name	Attribute	Default	Description
<dll>	*name*	Varies with plugin	Names a library file that implements a synchronization algorithm for a particular device type. For Apple devices running iAP1, you can use three synchronizers, which are implemented by the following libraries: **mss-ipodgeneric.so**, **mss-ipodpb.so**, and **mss-ipoduid.so**. For Apple devices running iAP2, you can use the synchronizer implemented by **mss-ipodiap2.so**. These synchronizers aren't included with the shipped version of **mm-sync**—to use them, you must purchase support for Apple devices by contacting your QNX Project Manager. Then, you can learn more about these synchronizers by reading *Working with iPods*, which is shipped with the QNX Apps and Media Interface for Apple iPod.

Related Links

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

The <Configuration>/<Database>/<Synchronization>/<PLSS> element

The **<Configuration>/<Database>/<Synchronization>/<PLSS>** element defines playlist session synchronizer (PLSS) settings related to resource usage and playlist entry matching. This element can also specify a configuration file for the **mmplaylist** library, which **mm-sync** uses to synchronize playlists.

The contents of the **<PLSS>** element are:

Tag name	Attribute	Default	Description
<MMPlaylistConfigFile>		`/etc/mm/mm-playlist.conf`	Names the configuration file for the **mmplaylist** library.

Tag name	Attribute	Default	Description
<Matching>	*ignore_case*	false	Specifies whether to ignore case when matching playlist entries. You must ensure that the ICU library is loaded to provide support for Unicode strings, or the matching may produce unexpected results. The matching will be performed based on the language code listed in the *lang_code* column of the **playlists** table row. You must use four-character language codes, such as the default 'en_CA'.
<UnresolvedEntryMask>		(empty)	Defines a POSIX regular expression (regex) for naming unresolved playlist entries you want synchronized to the **playlist_entries** table. Only one **UnresolvedEntryMask** can be specified. The default setting is empty, meaning no unresolved playlist entries get synchronized.

Identifying filenames of unresolved playlist entries to be synchronized

You can define complex text patterns using special characters and regex operators to support flexible filename matching of unresolved playlist entries. For example, to add unresolved playlist URLs for the HTTP, HTTPS, and FTP protocols to **playlist_entries**, set the **UnresolvedEntryMask** value to:

```
^(http(s?)|ftp)://
```

The special characters and operators supported for **<UnresolvedEntryMask>** are the same as those supported for **<SyncFileMask>**, which defines names of files to skip for synchronization.

Related Links

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Skipping files based on their names (p. 53)
The **<SyncFileMask>** element allows you to ignore files during synchronization based on a character string in the filename (including the file extension).

The <Configuration>/<Database>/<Prune> element

The **<Configuration>/<Database>/<Prune>** element defines values that influence how **mm-sync** monitors the database size and stops synchronization of metadata if the database gets too big.

The contents of this element are:

Tag name	Default	Description
<MaxDatabaseSize>	0	Sets the database size threshold (in kilobytes) for stopping synchronization. The database size can exceed this value by the amount of space needed to store information for the number of files specified in **<SyncInterval>**, which is an

Tag name	Default	Description
		indeterminate amount of space. So the threshold value serves as a size guideline but not a hard limit. If **mm-sync** checks the database size during synchronization and finds that the size has grown beyond this value, **mm-sync** stops any ongoing synchronization of the corresponding mediastore. A setting of 0 disables this size management; values greater than 0 enforce it.
<**SyncInterval>**	0	Sets the number of files added to the database by **mm-sync** between database size checks. Larger values means the size can exceed the **<MaxDatabaseSize>** threshold by a wider margin before **mm-sync** stops the synchronization. Smaller values mean more frequent size checks and hence, slower synchronizations. A setting of 0 disables size checking; values greater than 0 enforce it.

Related Links

The <Configuration> element (p. 42)
The **<Configuration>** element defines the device path used by **mm-sync** and contains the **<Database>** element, which controls the synchronization process behavior.

The <Configuration>/<Database> element (p. 42)
The **<Configuration>/<Database>** element defines some general database settings and the settings that control media synchronization and database size.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Setting synchronization thread priorities

You can set priorities for the foreground synchronization threads and for the merge thread. Adjusting thread priorities can have a significant impact on system performance.

The **<Priority>** element controls the priority of the foreground synchronization threads. Setting the element's value to 0 makes the foreground threads run at their default priority level, which matches that of the **mm-sync** service when it's started. Nonzero values assign absolute priorities.

Running the foreground threads at a priority level two lower than the default priority level may significantly reduce delays when changing tracks. For example, if **mm-sync** is running at priority 10, you should set the prioity of the foreground synchronization threads to 8: **<Priority>**8**</Priority>**.

The **<MergePriorityAdjust>** element controls the priority of the merge thread, which is a background synchronization thread that writes entries to the **files** table. By default, this thread runs at a priority one higher than that of the foreground synchronization threads. For example, if the foreground synchronization threads run at priority 10, the merge thread runs at priority 11.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Skipping files based on their names

The **<SyncFileMask>** element allows you to ignore files during synchronization based on a character string in the filename (including the file extension).

The default configuration is to not skip any files based on their name. There can be only one **<SyncFileMask>** element in the configuration file, but you can use regular expressions (regexes) to define complex text patterns, which gives you a lot of flexibility in specifying the files to skip in synchronizations.

Complex regex patterns slow down the files pass of synchronization.

To create the mask defining the character string that identifies the files to ignore during synchronization, use the following syntax:

Character	Meaning
\	Treat the next character as a literal
^	Begins with
$	Ends with

The **<SyncFileMask>** element supports POSIX regular expressions, so you can use AND (&) and OR (|) operators to create your mask.

Here are some simple masks to ignore files with names that:

begin with "."

<SyncFileMask>^\.</SyncFileMask>

The ".", which usually means any character, here means the "." (dot) character, because it's preceeded by a "\".

contain "w"

<SyncFileMask>w</SyncFileMask>

end with "roy"

<SyncFileMask>roy$</SyncFileMask>

begin with a "." (dot) character or end with ".mp3".

<SyncFileMask>(^\.)|(\.mp3$)</SyncFileMask>

match "Recycle Bin"

<SyncFileMask>^Recycle Bin$</SyncFileMask>

If you create a file mask with multiple operators, be sure that you don't configure **mm-sync** to ignore certain files that you want synchronized.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Filtering synchronization by file type (p. 55)
Client applications can filter the types of media files and playlists that get synchronized. The **<extensions>** element contains the **<library>** and **<playlists>** elements, which list the extensions that media files and playlists must have to be synchronized. Mediastore files with unlisted extensions don't get synchronized.

Filtering synchronization by file type

Client applications can filter the types of media files and playlists that get synchronized. The **<extensions>** element contains the **<library>** and **<playlists>** elements, which list the extensions that media files and playlists must have to be synchronized. Mediastore files with unlisted extensions don't get synchronized.

Both of these contained elements are required, and must list all extensions supported for that content type. This is done by defining an **<extension>** element for each supported file extension, within the **<library>** or **<playlists>** element.

The contents of the **<extensions>** element are:

Tag name	Attribute	Default	Description
<extensions>/<playlists>			Specifies which file extensions are used for playlists. For each extension you want to support, define a separate **<extension>** element.
<extensions>/<playlists>/<extension>	value	(none)	Names the extension of a playlist type to synchronize. Typically, the extension is a three- or four-letter abbreviation of the format, although longer extensions are accepted. The matching ignores case, so listing an extension of "m3u8" means playlists with the extension "M3U8" will also be synchronized.
<extensions>/<library>			Specifies which file extensions are used for media files. For each extension you want to support, define a separate **<extension>** element.
<extensions>/<library>/<extension>	value	(none)	Names the extension of a media file type to synchronize. Typically, the extension is a three- or four-letter abbreviation of the format, although longer extensions are accepted. The matching ignores case, so listing an extension of "mpeg4" in the configuration file means playlists with the extension "MPeg4" will also be synchronized.
	ftype	(none)	Names the default media type associated with the file extension. Acceptable values are `audio`, `video`, and `photo`. The ftype may change during the metadata pass of synchronization. For example, a file with an "mp4" extension can be an audio or video file. The customer must configure the file type to either `audio` or `video` to be used when **mm-sync** intially parses the file's information during the files pass. However, the *ftype* may change during the subsequent metadata pass if it's determined that the actual file type differs from the specified default type.

Suppose you want the database to hold photo metadata from compact bitmap (rasterized) file formats (and no audio or video metadata). You would then put the following in your configuration file:

```
<extensions>
    <library>
        <extension value="png" ftype="photo" />
        <extension value="jpg" ftype="photo" />
        <extension value="jpeg" ftype="photo" />
        <extension value="gif" ftype="photo" />
    </library>
</extensions>
```

Only media files with one of these four listed extensions will have their metadata extracted and uploaded to the database.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Skipping files based on their names (p. 53)
The **<SyncFileMask>** element allows you to ignore files during synchronization based on a character string in the filename (including the file extension).

Prescanning for nonmedia items

To prevent **mm-sync** from spending too much time searching through nonmedia items, you can configure parameters for the prescanning of folders. The prescanning determines which folders can be skipped from synchronization because they don't contain much (or any) media content.

<NonMediaItems>

The **<NonMediaItems>** element contains the elements that define the prescan parameters. At most one **<NonMediaItems>** tag may be specified in the configuration because this element controls the prescanning of all folders on all mediastores.

<MaxItems>

This element specifies the number of nonmedia items allowed in a folder before **mm-sync** considers it to be a system (i.e., nonmedia) folder and hence, doesn't synchronize its contents. Defining this limit prevents **mm-sync** from searching through sections of mediastore filesystems that contain primarily or exclusively nonmedia content.

By default, this tag is set to 0, which disables the limit on nonmedia items in a folder.

The *consecutive* attribute controls whether the nonmedia items must be consecutive within the folder listings to count towards the total number of nonmedia items found. For example, if *consecutive* is true and **<MaxItems>** is 50, then 50 consecutive nonmedia items must be found before **mm-sync** stops synchronizing the folder. Any media item found before this nonmedia items limit is reached causes the nonmedia items counter to be reset to 0.

Because the order of the folder listings is usually not guaranteed, enabling a consecutive limit can produce nondeterministic results. Hence, the attribute's default setting is false.

<PrescanLimit>

You can limit how many items **mm-sync** examines when prescanning a folder. Setting the **<PrescanLimit>** tag is useful for restricting the time spent reading a large folder (e.g., one with 20 000 items).

The default value is 0; this setting disables the limit on the items examined.

Prescan behavior

When enabled in the configuration, the prescan operation is done for each folder in the synchronization path. The prescan behavior is:

- Scan a folder until the limit on the number of folder items (**<PrescanLimit>**) is reached. All items—media and nonmedia—count towards this limit. If the corresponding tag is set to 0, keep scanning until all the folder items have been examined or the limit of nonmedia items is reached.
- When a nonmedia item is found, increment the corresponding counter. If the limit on nonmedia items (**<MaxItems>**) has been reached, stop examining the folder and don't synchronize it.
- If the *consecutive* attribute for this last tag is set to true, reset the counter of nonmedia items to 0 each time a media item is found.
- If the prescan finishes for a folder and the nonmedia item limit has not been exceeded, synchronize the folder.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Limiting the number of items read (p. 59)
You can limit the total number of items read from any folder or the entire mediastore. An excessively large number of items can make using the device very slow and can cause poor response times for synchronization operations such as changing priority folders or determining mediastore changes.

Mediastore filesystem traversal (p. 13)
At each synchronization pass, **mm-sync** traverses the mediastore filesystem to extract and upload media information into the mediastore's database. The section of the filesystem tree that is synchronized depends on the user-specified path. If a blank path is given and the recursive option is set, the entire filesystem is synchronized. Otherwise, only the files and folders named by the path are synchronized.

Limiting the number of items read

You can limit the total number of items read from any folder or the entire mediastore. An excessively large number of items can make using the device very slow and can cause poor response times for synchronization operations such as changing priority folders or determining mediastore changes.

<MaxFolderItems>

The **<MaxFolderItems>** element controls the maximum number of items read from a folder. The configuration file can contain only one **<MaxFolderItems>** tag; its value will apply to every folder on every mediastore. Currently, only the block filesystem (BFS) mediastore synchronizer (**bfsrecurse**) uses the value in the **<MaxFolderItems>** element; other synchronizers ignore it. The default setting for **<MaxFolderItems>** is 0, meaning there's no default limit on the number of items read from a folder.

To keep the count consistent, independently of the type of items in the folder and of their contents, **mm-sync** calculates the number of items in a folder before performing any filtering based on the **<SyncFileMask>** setting. Because "." and ".." are always ignored, **mm-sync** compensates for these files when it calculates the number of items in a folder. When a synchronization reaches the limit set by **<MaxFolderItems>**, **mm-sync** stops synchronizing the folder and delivers the MMSYNC_SYNC_ERROR_FOLDER_LIMIT event.

When you configure a limit on the number of items read from any one folder by setting **<MaxFolderItems>** to a nonzero value, the filesystem `readdir()` operation determines which folder items **mm-sync** sees. The **mm-sync** service doesn't see items beyond the **<MaxFolderItems>** limit and doesn't consider them when checking for folder changes. These unseen items may include media files, playlist files, other folders, and files that have been filtered out or that, by definition, **mm-sync** doesn't handle (such as **.xls** files). Adding or deleting items from a folder on a mediastore affects the presentation of the items to **mm-sync** (depending on the operation of the *readdir()* function) and may cause unexpected changes to the database for the mediastore.

<MaxMediaStoreItemsConfiguration>

The **<MaxMediaStoreItemsConfiguration>** element and its contained **<MaxMediaStoreItems>** elements control the maximum number of items read from mediastores. There can be only one **<MaxMediaStoreItemsConfiguration>** element, but it can have multiple **<MaxMediaStoreItems>** elements to set distinct limits on the number of items read for different mediastores. Each **<MaxMediaStoreItems>** element must have a `mediastore` attribute set to the mediastore path and the element value set to the limit on the number items to read. You can use wildcards in the mediastore path. For example, the configuration element **<MaxMediaStoreItems mediastore="/fs/usb*">100</MaxMediaStoreItems>** means the devices located at **/fs/usb0** and **/fs/usb1** both have a maximum of 100 items read.

Currently, only the following mediastore synchronizers use the values in the **<MaxMediaStoreItemsConfiguration>** and **<MaxMediaStoreItems>** elements:

- **audiocd**
- **bfsrecurse**

During synchronization, **mm-sync** walks the device's directory structure, starting from the root folder in the synchronization path. For each folder it examines, **mm-sync** first calculates the number of items, ignoring the "." and ".." filesystem entries, and then performs any filtering based on the **<SyncFileMask>** setting.

If **<MaxMediaStoreItems>** is 0 for the mediastore being synchronized or if there's no element whose `mediastore` attribute matches the path of the mediastore, **mm-sync** will read an unlimited number of items from that device. If there's a matching path in a **<MaxMediaStoreItems>** element that has a nonzero value, the synchronization process counts how many items have been read from all folders traversed so far, and then increments this count for each item read, whether the item gets synchronized or not. When the number of items read from the mediastore reaches the configured limit, **mm-sync** delivers the MMSYNC_SYNC_ERROR_LIB_LIMIT event.

The **mm-sync** process doesn't see items beyond the **<MaxMediaStoreItems>** limit, meaning that some content—media files, playlist files, folders, and filtered and unsupported files—may not be read and hence may not be synchronized to the database for the mediastore.

Items synchronized vs. items read

The number of items synchronized will be at most equal to the limit on how many items can be read. If you haven't specified a file mask to ignore certain files, then the number of items synchronized will match the maximum number of items read.

If you have defined a file mask to skip files in the synchronization based on their names, then the number of items synchronized will be equal to the limit on items read minus the number of items that matched the file mask. For example, suppose you set **<MaxFolderItems>** to 100 and **<SyncFileMask>** to "^\." (to ignore files beginning with "."). If a mediastore folder contains 150 items and if 20 of the first 100 items are files, subfolders, or playlists that match the file mask, then the number of items synchronized for the folder will be 100 - 20 = 80. Again, **mm-sync** doesn't consider items beyond the limit on the number of items read, so in this case, the last 50 folder items are never considered for synchronization.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Prescanning for nonmedia items (p. 57)
To prevent **mm-sync** from spending too much time searching through nonmedia items, you can configure parameters for the prescanning of folders. The prescanning determines which folders can be skipped from synchronization because they don't contain much (or any) media content.

Mediastore filesystem traversal (p. 13)
At each synchronization pass, **mm-sync** traverses the mediastore filesystem to extract and upload media information into the mediastore's database. The section of the filesystem tree that is synchronized depends on the user-specified path. If a blank path is given and the recursive option is set, the entire filesystem is synchronized. Otherwise, only the files and folders named by the path are synchronized.

Maintaining constant IDs for updated files and playlists

The **<ChangedFilesHaveConstantId>** element in the configuration file and the `dynamic_folder` option in the command for starting a synchronization both force **mm-sync** to keep the same IDs in the database for media and playlist files that have been modified.

By default, during the files pass of synchronization, **mm-sync** treats any media or playlist file whose size or date has changed as new and synchronizes it accordingly, adding a table entry with a new file ID. For new media files, it's a **files** table entry with a unique *fid* value. For new playlist files, it's a **playlists** table entry with a unique *plid* value. When either the **<ChangedFilesHaveConstantId>** configuration setting or the `dynamic_folder` option for a particular synchronization is enabled, the files pass does *not* consider media files and playlists with changes in size or modification date as new, but maintains their file IDs and sets their *accurate* fields to 0 to indicate that the file metadata might have changed.

Thus, if you configure **mm-sync** to maintain constant file IDs for changed media or playlist files, you must check a file's *accurate* field before you can use its metadata. If the *accurate* field is 0, then the file requires a metadata or playlist synchronization pass.

This setting slows down the files pass of synchronization when the filesystem has changed, but is necessary for legacy media applications that aren't designed to accommodate the file ID changing when a file is modified.

For media files not successfully synchronized during the metadata pass or for playlist files not successfully synchronized during the playlist pass, their *accurate* field will be set to 0, regardless of the **<ChangedFilesHaveConstantId>** or `dynamic_folder` setting.

Playlist synchronization behavior

Playlist synchronization differs from media file synchronization in the following ways:

- The playlist pass ensures all playlists are synchronized, except for those already marked as accurate, which are skipped from synchronization.
- For playlists marked as accurate, the playlist pass validates their entries against the corresponding media files in the **files** table, checking if any files have been added or removed.
- If the media file referred to by a playlist entry has been modified, the playlist entry is marked as not accurate; **mm-sync** will reparse that entry on the next playlist synchronization.

Related Links

Configuration file contents (p. 42)
The configuration file contains a hierarchy of XML elements that control all synchronization features. Your own file must follow the prescribed structure so the synchronizer service can properly parse your file, but you need to include only those elements that specify the properties you want to use.

The <Configuration>/<Database>/<Synchronization> element (p. 43)
The **<Configuration>/<Database>/<Synchronization>** element defines the settings that control synchronization, including the optimization of system resources, the file types to include or exclude, the mapping of metadata provider fields to database fields, and policies for media data and playlists.

Chapter 5
Multimedia Synchronizer API

The multimedia synchronizer API exposes the constants, data types, and functions that client applications can use to start synchronizing media content, monitor synchronization progress, and interpret any errors reported by the **mm-sync** service.

The API consists of five sections:

Client interface

Defines the flags, data types, and functions for managing **mm-sync** connections and for controlling and monitoring synchronizations.

Configuration constants

Specify values used to configure synchronization operations.

Media file categories

Support a mapping of the media file type to the database tables written during synchronization and also allow for filtering query results by file type.

Event interface

Defines functions for processing **mm-sync** events as well as data types for listing the event types and storing the data of specific event types.

Error information

Includes a listing of error types and a structure that holds information about individual errors.

Client interface

The **mm-sync** service provides a client interface that allows applications to connect to the service, start and monitor synchronizations, and set debugging levels.

The client interface defines functions for synchronizing specific media content on a device and for suspending, cancelling, or resuming a synchronization. This API section also defines synchronization option flags and data types used as function parameters.

Before you can synchronize any content, you must connect to **mm-sync** by calling *mm_sync_connect()*. This function returns a connection handle (as an **mmsync_hdl_t** value) that must be used in subsequent API calls to refer to the same connection. For example, you can start synchronizations by calling *mm_sync_start()*, passing in that connection handle along with the mountpoint of the device and the path containing the content that you want to synchronize.

When you're finished synchronizing media content, you can disconnect from **mm-sync** by calling *mm_sync_disconnect()*.

There's also a function for setting the debugging and verbosity levels (*mm_sync_debug_set()*) so you can see useful troubleshooting information in the **stderr** file stream and **sloginfo** logging utility.

Client interface constants

Constants for enabling synchronization options, making some function calls blocking, and limiting debugging levels that can be set by clients

Flags for enabling synchronization options:

Synopsis:

```
#include <mmsync/interface.h>
```

Defines:

#define MMSYNC_OPTION_PASS_FILES (0x00000001)

Perform files pass of synchronization.

#define MMSYNC_OPTION_PASS_METADATA (0x00000002)

Perform metadata pass of synchronization.

#define MMSYNC_OPTION_PASS_PLAYLISTS (0x00000004)

Perform playlist pass of synchronization.

#define MMSYNC_OPTION_PASS_ALL (MMSYNC_OPTION_PASS_FILES + \ MMSYNC_OPTION_PASS_METADATA + \ MMSYNC_OPTION_PASS_PLAYLISTS)

Perform all three passes of synchronization.

#define MMSYNC_OPTION_REPAIR (0x00000400)

Repair database inconsistencies in folder information.

#define MMSYNC_OPTION_VERIFY (0x00000800)

Verify data consistency in folder information.

#define MMSYNC_OPTION_CANCEL_CURRENT (0x00002000)

Cancel any ongoing synchronization when a new one is started.

#define MMSYNC_OPTION_RECURSIVE (0x00004000)

Synchronize contents of subfolders within the folder named in the path.

#define MMSYNC_OPTION_SKIPNONMEDIACHECK (0x00010000)

Skip check of nonmedia files.

Debugging flags

Synopsis:

```
#include <mmsync/interface.h>
```

Defines:

#define MMSYNC_DEBUG_LOG_EMIT_TO_STDERR (0x01)

Indicates log messages are sent to **stderr** (in addition to **slog**).

Controls for blocking on function calls and for limiting verbosity and debugging:

Synopsis:

```
#include <mmsync/mmsyncclient.h>
```

Defines:

#define MM_SYNC_SUSPEND_FLAGS_WAIT 0x00000001

Force *mm_sync_suspend()* to block until synchronization thread is suspended.

#define MMSYNC_MAX_VERBOSE 10

Restrict verbosity setting for *mm_sync_debug_set()* to maximum of 10.

#define MMSYNC_MAX_DEBUG 0xff

Allow unlimited debugging setting for *mm_sync_debug_set()*.

Flag for indicating synchronization thread suspension:

Synopsis:

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

Defines:

#define MM_SYNC_THREAD_IS_SUSPENDED 0x00000001

Indicate current synchronization operation has a suspended thread.

Library:

mmsyncclient

mm_sync_cancel()

Cancel a synchronization

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_cancel( mmsync_hdl_t *hdl,
                    unsigned op_id,
                    uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

op_id

The operation ID of the synchronization to cancel.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Description:

Cancel a synchronization. You can do this if a device is no longer of interest or if playback is requested from a device that's being synchronized and that has problems with simultaneous playback and synchronization.

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_control() (p. 69)
Send commands to a synchronization in progress

mm_sync_resume() (p. 74)
Resume a suspended synchronization

mm_sync_start() (p. 75)
Start a synchronization

mm_sync_suspend() (p. 84)
Suspend a synchronization

mm_sync_connect()

Connect to ***mm-sync*** *and obtain a handle*

Synopsis:

```
#include <mmsync/mmsyncclient.h>

mmsync_hdl_t* mm_sync_connect( const char *filename,
                               uint32_t flags )
```

Arguments:

filename

The path to the **mm-sync** device name.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Description:

Connect to **mm-sync** and obtain a handle. The *filename* argument must contain the path of the device object used for synchronizations. The default device is **/dev/mmsync**, but the path you provide to this function must match what you gave to **mm-sync** at startup.

Returns:

An **mm-sync** connection handle on success, NULL on failure.

Related Links

mm_sync_disconnect() (p. 73)

Disconnect from ***mm-sync***

mm_sync_control()

Send commands to a synchronization in progress

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_control( mmsync_hdl_t *hdl,
                     uint32_t op_id,
                     strm_dict_t *extended_options,
                     uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

op_id

The operation ID of the synchronization being controlled.

extended_options

A set of key/value pairs containing synchronization control commands, formatted as follows:

```
key1=value1,key2=value2,key3=value3,...
```

Key	Value	Description
`action`	Currently, only one action is supported: `priority_folder_set`	The action to perform on the synchronization.
`folderid`	An integer storing the ID of a mediastore folder.	The folder the action is performed on. Either this field or *folder_path* must be defined for actions such as `priority_folder_set` that affect a particular folder.
`folder_path`	A string storing the path of a mediastore folder. The path is relative to the mediastore's filesystem (e.g., "/" refers to the mediastore's root folder).	The folder the action is performed on. Either this field or *folderid* must be defined for actions such as `priority_folder_set` that affect a particular folder.
`dynamic_folder`	enable | disable	Enable or disable the dynamic setting for the folder referred to by *folderid* or *folder_path*. This option applies to the `priority_folder_set` action.

Key	Value	Description
		When this setting is enabled, the *fids* for files in this folder will remain constant. The setting is nonrecursive, so the only files affected are those in the top-level folder in the synchronization path named in the operation *op_id*; files in subfolders aren't affected. For information on how this setting impacts synchronization, see "*Maintaining constant IDs for updated files and playlists* (p. 61)".

Action	Description
`priority_folder_set`	Initiates a priority folder synchronization. Requires one of the *folderid* and *folder_path* key/value pairs. You can also define the *dynamic_folder* key to enable or disable the dynamic folder setting.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Description:

Send commands to a synchronization in progress. Currently, the only command supported is **priority_folder_set**, which initiates a priority folder synchronization.

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_cancel() (p. 67)
Cancel a synchronization

mm_sync_resume() (p. 74)
Resume a suspended synchronization

mm_sync_start() (p. 75)
Start a synchronization

mm_sync_suspend() (p. 84)
Suspend a synchronization

mm_sync_debug_get()

Get the logging verbosity level and debugging flags

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_debug_get( mmsync_hdl_t *hdl,
                       uint8_t *verbose,
                       uint8_t *debug )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

verbose

A pointer to the storage for the verbosity level.

debug

A pointer to the storage for the debugging flags.

Library:

mmsyncclient

Returns:

0 on success, -1 on failure.

mm_sync_debug_set()

Set the logging verbosity level and debugging flags

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_debug_set( mmsync_hdl_t *hdl,
                       uint8_t verbose,
                       uint8_t debug )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

verbose

The new verbosity level.

debug

The new debugging flags (as a bitfield of MMSYNC_DEBUG_* constants). This setting overrides the previous flags.

Library:

mmsyncclient

Description:

Set the logging verbosity level and debugging flags. These settings can be between 0 (to turn off the feature) and the maximum limits defined by MMSYNC_MAX_VERBOSE and MMSYNC_MAX_DEBUG. Values higher than these limits cause the function to fail.

Returns:

0 on success, -1 on failure.

mm_sync_disconnect()

*Disconnect from **mm-sync***

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_disconnect( mmsync_hdl_t *hdl )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

Library:

mmsyncclient

Description:

Disconnect from **mm-sync**. After calling this function, don't use the **mm-sync** connection handle.

Returns:

0 on success, -1 on failure (i.e., some resources couldn't be fully released).

Related Links

mm_sync_connect() (p. 68)

*Connect to **mm-sync** and obtain a handle*

mm_sync_resume()

Resume a suspended synchronization

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_resume( mmsync_hdl_t *hdl,
                    unsigned op_id,
                    uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

op_id

The operation ID of the suspended synchronization to resume.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_cancel() (p. 67)
Cancel a synchronization

mm_sync_control() (p. 69)
Send commands to a synchronization in progress

mm_sync_start() (p. 75)
Start a synchronization

mm_sync_suspend() (p. 84)
Suspend a synchronization

mm_sync_start()

Start a synchronization

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_start( mmsync_hdl_t *hdl,
                   const char *db,
                   const char *mountpoint,
                   const char *syncpath,
                   uint32_t options,
                   const strm_dict_t *extended_options )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

db

The device path of the database to store the synchronized content.

mountpoint

The mountpoint of the mediastore to synchronize.

syncpath

The relative path on the mediastore of the files or folders to synchronize.

options

The following synchronization options apply:

MMSYNC_OPTION_PASS_FILES

Perform the files pass.

MMSYNC_OPTION_PASS_METADATA

Perform the metadata pass.

MMSYNC_OPTION_PASS_PLAYLISTS

Perform the playlist pass.

MMSYNC_OPTION_PASS_ALL

Perform all three passes.

MMSYNC_OPTION_CANCEL_CURRENT

Cancel any synchronization in progress on the device before starting the new synchronization.

MMSYNC_OPTION_RECURSIVE

After synchronizing the root folder in the specified path, synchronize its subfolders.

extended_options

A set of key/value pairs containing extended synchronization options, formatted as follows:

```
key1=value1,key2=value2,key3=value3,...
```

This parameter can be NULL.

Key	Value	Description
`use_synchronizer`	"*mss_name*" (a supported synchronizer, as a string in quotes; e.g., `"dvdaudio"`)	Use the specified synchronizer if it supports the current operation; otherwise, do nothing.
`force_synchronizer`	"*mss_name*" (a supported synchronizer, as a string in quotes; e.g., `"dvdvideo"`)	Force the use of the specified synchronizer, whether or not it supports the current operation.
`dynamic_folder`	enable | disable	Enable or disable the dynamic setting for the folder specified in *syncpath*. The *fids* for files in this folder will remain constant while this setting is enabled. The setting is nonrecursive, so the only files affected are those in the top-level folder given in *syncpath*; files in subfolders aren't affected. For information on how this setting impacts synchronization, see "*Maintaining constant IDs for updated files and playlists* (p. 61)".

`metadata_keys`	The metadata fields to be read, as name-value pairs separated by semi-colons:
	
	`md_title_name=Poltergeist;`
	`md_title_genre=Horror;`
	`md_title_album=`
	`UnleashTheDemons;`
	`md_title_artist=Mr_X`
		Retrieve only the listed metadata fields. This setting affects only directed synchronizations in which `dynamic_folder` is enabled.
		
		When you define `metadata_keys`, the **libmd** library isn't used for metadata extraction; instead, **mm-sync** sets the metadata fields to the values listed in this option. Note that you must provide values for each field that you list.

Library:

mmsyncclient

Description:

Start synchronizing the media content contained in *syncpath*. This path is relative within the filesystem of the mediastore located at *mountpoint*. The media content is synchronized to the database with the device path in *db*. The **mm-sync** process synchronizes content in a dedicated thread, so this function call returns before the synchronization starts.

All path arguments must contain valid locations in a locally accessible filesystem. The *syncpath* argument is flexible, allowing you to specify these scopes:

- the entire mediastore
- a section of the mediastore filesystem, based on a root folder
- an individual file, which may be a media file (i.e., a track) or a playlist

When the synchronization path refers to a folder, you must terminate it with a slash (**/**). To make **mm-sync** look in subfolders, set the MMSYNC_OPTION_RECURSIVE flag in *options*. When the path is an individual file, you must set both the MMSYNC_OPTION_PASS_FILES and MMSYNC_OPTION_PASS_METADATA flags for media files and set the MMSYNC_OPTION_PASS_PLAYLISTS flag for playlists.

Some mediastores don't support synchronizations of specific folders or files. For these mediastores, you must synchronize all their content by specifying a path of `"/"`.

Returns:

Values greater than 0 refer to the synchronization operation ID, on success. -1 is returned on failure.

Related Links

Full, directed, and file synchronizations (p. 17)
The multimedia synchronizer doesn't provide separate controls for synchronizing an entire mediastore versus certain folders or files. You use the same function call to synchronize content whether it's a full, recursive synchronization of all the mediastore content or of only a folder, file, or playlist.

Synchronizer selection (p. 10)
The **mm-sync** service provides many synchronizers designed for various media and storage devices. Some synchronizers can extract the metadata from a certain device, media type, or playlist better than other synchronizers. When **mm-sync** receives a synchronization request, it selects the best synchronizer to use for the content being synchronized and for the device and media type.

mm_sync_cancel() (p. 67)
Cancel a synchronization

mm_sync_control() (p. 69)
Send commands to a synchronization in progress

mm_sync_resume() (p. 74)
Resume a suspended synchronization

mm_sync_suspend() (p. 84)
Suspend a synchronization

mm_sync_status_get()

Get the statuses of active synchronizations

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_status_get( mmsync_hdl_t *hdl,
                        mmsync_status_t *status,
                        size_t max_num_status,
                        uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

status

A pointer to an array of structures to hold the statuses.

max_num_status

The array size.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Description:

Get the statuses of active synchronizations, including those started before your application connected to **mm-sync**.

This function writes the statuses of individual synchronizations into separate **mmsync_status_t** structures. Note that *max_num_status* is the maximum number of structures that **mm-sync** will write to, not the size of the **mmsync_status_t** data type. So, you must provide a buffer of `sizeof(mmsync_status_t) * max_num_status` bytes in *status*.

If you want to get information about all active synchronizations, you can first call *mm_sync_status_get()* with *max_num_status* to 0 and *status* to NULL. The function will return the total number of synchronizations in progress or pending, which you can use to calculate the amount of buffer space needed. You can then call this function again and **mm-sync** will write the statuses of all synchronizations into the sufficiently large buffer.

Returns:

On success, a value greater than 0 indicating the number of synchronizations in progress or pending. This value may be greater than *max_num_status* but only the minimum of this return value or *max_num_status* statuses are available. On failure, -1 is returned.

Related Links

mmsync_status_t (p. 86)
The synchronization status of a single mediastore

mm_sync_status_get_bydbname() (p. 80)
Get the status of a synchronization based on the database name

mm_sync_status_get_byid() (p. 81)
Get the status of a synchronization based on the operation ID

mm_sync_status_get_dbname() (p. 82)
Get the name of the database used in a specific synchronization

mm_sync_status_get_bydbname()

Get the status of a synchronization based on the database name

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_status_get_bydbname( mmsync_hdl_t *hdl,
                                 mmsync_status_t *status,
                                 const char *dbname,
                                 uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

status

A structure to store the status in.

dbname

The name of the database for which the status is being retrieved.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Returns:

On success, a value greater than 0 indicating the number of results. When no database with the given name is found, the function returns 0 because there are no results; this isn't considered an error case. On failure, -1 is returned.

Related Links

mmsync_status_t (p. 86)
The synchronization status of a single mediastore

mm_sync_status_get() (p. 78)
Get the statuses of active synchronizations

mm_sync_status_get_byid() (p. 81)
Get the status of a synchronization based on the operation ID

mm_sync_status_get_dbname() (p. 82)
Get the name of the database used in a specific synchronization

mm_sync_status_get_byid()

Get the status of a synchronization based on the operation ID

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_status_get_byid( mmsync_hdl_t *hdl,
                             mmsync_status_t *status,
                             uint32_t id,
                             uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

status

A structure to store the status in.

id

The operation ID of the synchronization for which the status is being retrieved.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Returns:

On success, a value greater than 0 indicating the number of results. When no synchronization with the given operation ID is found, the function returns 0 because there are no results; this isn't considered an error case. On failure, -1 is returned.

Related Links

mmsync_status_t (p. 86)
The synchronization status of a single mediastore

mm_sync_status_get() (p. 78)
Get the statuses of active synchronizations

mm_sync_status_get_bydbname() (p. 80)
Get the status of a synchronization based on the database name

mm_sync_status_get_dbname() (p. 82)
Get the name of the database used in a specific synchronization

mm_sync_status_get_dbname()

Get the name of the database used in a specific synchronization

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_status_get_dbname( mmsync_hdl_t *hdl,
                               uint32_t id,
                               char *dbname,
                               size_t dbname_sz,
                               uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

id

The operation ID of the synchronization for which the database name is being retrieved.

dbname

A buffer to store the database name.

dbname_sz

The buffer size.

flags

Reserved for future use, must be 0.

Library:

mmsyncclient

Description:

Get the name of the database used in the synchronization with the operation ID matching *id*. The database name is copied into the *dbname* buffer, with at most *dbname_sz* bytes being written. If there's no active synchronization with this operation ID (e.g., the synchronization has already finished), nothing is written to the buffer.

If you need to know how much memory to allocate for the buffer, call *mm_sync_status_get_dbname()* with *dbname* set to NULL and *dbname_sz* set to 0. The function then returns the number of bytes needed to store the database name.

Returns:

On success, the required buffer size for storing the database name. On failure, -1 is returned.

Related Links

mm_sync_status_get() (p. 78)
Get the statuses of active synchronizations

mm_sync_status_get_bydbname() (p. 80)
Get the status of a synchronization based on the database name

mm_sync_status_get_byid() (p. 81)
Get the status of a synchronization based on the operation ID

mm_sync_suspend()

Suspend a synchronization

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_suspend( mmsync_hdl_t *hdl,
                     unsigned op_id,
                     uint32_t flags )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

op_id

The operation ID of the synchronization to suspend.

flags

Use MM_SYNC_SUSPEND_FLAGS_WAIT to block until the synchronization thread has been suspended.

Library:

mmsyncclient

Description:

Suspend a synchronization. You can do this to relieve contention on a device for a temporary purpose, such as accessing a file or starting playback quickly. This differs from cancelling a synchronization, which can be done if it's no longer needed or it begins to negatively impact performance.

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_cancel() (p. 67)
Cancel a synchronization

mm_sync_control() (p. 69)
Send commands to a synchronization in progress

mm_sync_resume() (p. 74)
Resume a suspended synchronization

mm_sync_start() (p. 75)
Start a synchronization

mmsync_hdl_t

*The **mm-sync** connection handle type*

Synopsis:

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

typedef struct _mmsync_hdl mmsync_hdl_t;
```

Library:

mmsyncclient

Description:

The **mmsync_hdl_t** structure is a private data type representing an **mm-sync** connection handle.

The *mm_sync_connect()* function returns a connection handle when a connection was successfully established. Your application must use this handle to access the same connection in all synchronization management API calls. After calling *mm_sync_disconnect()* to terminate the connection, you must not use the handle anymore.

Related Links

mm_sync_connect() (p. 68)

*Connect to **mm-sync** and obtain a handle*

mm_sync_disconnect() (p. 73)

*Disconnect from **mm-sync***

mmsync_status_t

The synchronization status of a single mediastore

Synopsis:

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

typedef struct s_mmsync_status {
    uint32_t operation_id;
    uint16_t passes_done;
    uint16_t current_pass;
    uint16_t passes_to_do;
    uint16_t reserved[1];
    uint32_t flags;
} mmsync_status_t;
```

Data:

uint32_t operation_id

The synchronization operation ID.

uint16_t passes_done

Flags indicating which synchronization passes have been completed.

uint16_t current_pass

Flag indicating which pass, if any, is in progress.

uint16_t passes_to_do

Flags indicating the synchronization passes that have not yet been started.

uint16_t reserved

Packing element.

uint32_t flags

Operation status flags (currently, only MM_SYNC_THREAD_IS_SUSPENDED is supported).

Library:

mmsyncclient

Description:

The synchronization status of a single mediastore.

For the *mm_sync_status_get_bydbname()* and *mm_sync_status_get_byid()* functions, you must provide one **mmsync_status_t** object as an input/output argument, to give these functions a space to write the status results.

For *mm_sync_status_get()*, you must pass in an array of **mmsync_status_t** objects as well as the array size in the function call, because this function returns the status of all active synchronizations (or as many statuses as can be stored in the array).

Related Links

mm_sync_status_get() (p. 78)
Get the statuses of active synchronizations

mm_sync_status_get_bydbname() (p. 80)
Get the status of a synchronization based on the database name

mm_sync_status_get_byid() (p. 81)
Get the status of a synchronization based on the operation ID

Configuration settings

The configuration settings define values used by **mm-sync** in controlling synchronization threads, reporting events, managing databases, limiting which media files get synchronized, and applying policies to certain components.

Configuration constants

*Constants for defining **mm-sync** configuration values*

Synopsis:

```
#include <mmsync/config.h>
```

General configuration settings:

#define CONF_DFLT_DEVICE_PATH "/dev/mmsync"

Default device path of **mm-sync** resource manager. All control contexts are created under this path.

#define CONF_DFLT_VERBOSITY_LEVEL 2

Default verbosity level for **mm-sync** initialization log messages.

Event settings:

#define CONF_NTFY_QUEUE_MAX (80)

Maximum number of event notifications that can be queued at a time.

#define CONF_DFLT_SYNC_FOLDER_EVENTS 2

Default emission control of folder synchronization events, which is `trimonly`.

#define CONF_EVENT_NUM_PL_ENTRIES_SYNC (0)

Number of playlist entries that can be updated before the MMSYNC_EVENT_PLAYLIST_ENTRIES_UPDATE event is sent.

#define CONF_DFLT_SYNC_MAX_FIRST_FID (1)

Default maximum number of MS_SYNC_FIRST_EXISTING_FID events sent during synchronization.

Thread resource management settings:

#define CONF_DFLT_SYNC_BUFFER (250)

Default maximum number of synchronization records that can be stored in buffers shared between foreground and background synchronization threads.

#define CONF_DFLT_SYNC_THREADS_MAX (8)

Default maximum number of foreground synchronization threads permitted to run at a time.

#define CONF_DFLT_SYNC_THREAD_PRIORITY 0

Default synchronization thread priority (0 means priority is inherited from main **mm-sync** thread).

#define CONF_DFLT_MERGE_THREAD_PRIORITY_ADJ 1

Default priority adjustment for merge thread. This value is added to the original synchronization thread priority, so it's a relative value.

Limits on files that get synchronized:

#define CONF_DFLT_SYNC_MAX_RECURSE (8)

Default maximum directory depth to recursively visit when synchronizing a mediastore. This setting also applies to priority folders.

#define CONF_MAX_MAXMEDIASTOREITEMS (100)

Maximum number of distinct mediastores for which you can define a limit on the number of media items read. This setting basically restricts how many **<MaxMediaStoreItems>** tags will be read from the configuration file.

#define CONF_DFLT_MEDIAITEMS_MAXITEMS (0)

Default maximum number of media items that can be synchronized in a folder before **mm-sync** skips the remaining items and moves onto another folder. This limit excludes "." and ".." filesystem entries but includes any items whose filenames match the name pattern for files to skip from synchronization (which is specified in the **<SyncFileMask>** tag).

Here, *media items* refers to media files as defined in the configuration file. Individual tracks, playlists, and folders can be considered media items.

When this setting is 0, all media items in a folder get synchronized. If the number of folder entries is greater than this configured limit, the subset of items that gets synchronized depends on the system.

#define CONF_DFLT_NONMEDIAITEMS_MAXITEMS (0)

Default maximum number of nonmedia items allowed in a folder before the folder is determined to not contain media items (0 means unlimited).

#define CONF_DFLT_NONMEDIAITEMS_PRESCANLIMIT (0)

Default maximum number of items to examine in a folder when prescanning for nonmedia content. All items—media and nonmedia—count towards the prescan limit.

When this setting is 0, **mm-sync** keeps scanning the folder until all of its items have been examined or the limit of nonmedia items is reached.

#define CONF_DFLT_MEDIAITEMS_SIZEMIN (0)

Minimum size that a discovered file must have to be treated as a new media file during synchronization (0 means any size).

Playlist limits

#define CONF_MAX_PLAYLIST_LINES 5000

Maximum number of lines that can be read from a playlist file. Each line names one playlist entry, so this setting limits the allowable playlist length. This value is used by some playlist session synchronizers (PLSSes).

Limits on lengths of string parameters:

#define CONF_MAX_PATH 4000

Maximum allowable length of any filename (including its path) given as a synchronization parameter.

#define CONF_MAX_SQL 8000

Maximum size of any message sent to the database resource manager; the message could be an SQL statement or something else.

#define CONF_MAX_EXT_LEN (30)

Maximum length of a file extension that can be specified in a filename given to **mm-sync**.

#define CONF_MAX_EXTS 200

Maximum number of distinct file extensions that can be synchronized. This setting is applied separately for the extensions of media files and the extensions of playlists.

#define CONF_MAX_MS_NAME_LEN (128)

Maximum allowable length for a volume name referring to an attached mediastore. The length of the volume name passed into **mm-sync** by the client can't exceed this configured limit; otherwise, the synchronization fails.

#define CONF_MAX_MSSNAME_LEN (20)

Maximum length of a mediastore synchronizer name that can be given in extended options to *mm_sync_start()*.

Database settings:

#define CONF_DFLT_MAX_DATABASE_SIZE 0

Default maximum size for the database, in kilobytes. If the database exceeds this size, the synchronization aborts. A size of 0 means no limit on the database size.

#define CONF_DFLT_SYNC_INTERVAL 0

Number of files to synchronize before checking the database size. Larger values mean faster synchronizations but also that the maximum database size can be exceeded by a greater margin before the synchronization aborts. If this setting is 0, **mm-sync** doesn't check the database size during synchronization.

#define CONF_DFLT_DATABASE_TIMEOUT_MS (0)

Default timeout value for database accesses, in milliseconds.

#define CONF_UTF8_CHAR_SIZE (4)

Number of characters used in specifying maximum string lengths for metadata. This value is used for copying metadata to the database.

#define CONF_MAX_METADATA_CHARS (256)

Size limit of metadata strings written to the database. This limit is the number of characters in the metadata strings copied from metadata providers into database fields.

#define CONF_DFLT_TYPE_CHARS (1)

Default type to use for truncating the metadata. The type can be either characters (the default) or bytes.

Library:

mmsyncclient

Media file categories

The media file categories provide metadata that **mm-sync** uses to determine which database tables and fields to populate based on a media file's extension. In the API, you can use these categories to filter database query results based on file contents (i.e., audio, video, or image data).

mm_ftypes_t

Media file categories

Synopsis:

```
#include <mmsync/interface.h>

typedef enum mm_ftypes_e {
    FTYPE_UNKNOWN = 0,
    FTYPE_AUDIO = 1,
    FTYPE_VIDEO = 2,
    FTYPE_RESERVED1 = 3,
    FTYPE_PHOTO = 4,
    FTYPE_DEVICE = 5,
    FTYPE_MAX
} mm_ftypes_t;
```

Data:

FTYPE_UNKNOWN

Unknown media file category.

FTYPE_AUDIO

Audio file.

FTYPE_VIDEO

Video file.

FTYPE_RESERVED1

Reserved for future use.

FTYPE_PHOTO

Photo file.

FTYPE_DEVICE

POSIX filesystem device.

FTYPE_MAX

End-of-list identifier.

Library:

mmsyncclient

Event interface

The **mm-sync** API defines general event categories and specific event types as well as structures for holding information returned for particular event types. This information can include the synchronization operation ID, timestamps, and details on the database file entries that were added or modified.

All events have an associated **mmsync_event_t** structure, which is returned by *mm_sync_events_get()*. The **mmsync_event_t** structure contains the event type, its size, and a field to access additional data. The data field points to the contents of another structure that holds extra information specific to the event's type.

mm_sync_events_get()

Get the next queued ***mm-sync*** *event*

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_events_get( mmsync_hdl_t *hdl,
                        mmsync_event_t **mmsync_event )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

mmsync_event

A pointer to the event.

Library:

mmsyncclient

Description:

Get the next queued **mm-sync** event. The event returned is stored in the **mmsync_hdl_t** object passed into the function. The client must not delete this event by passing it to *free()*. When *mm_sync_events_get()* is called on an **mmsync_hdl_t** object, it invalidates the previous **mmsync_event_t** returned. Clients that want to keep the old event should copy it before calling *mm_sync_events_get()* a second time.

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_events_register() (p. 97)

Register or unregister for ***mm-sync*** *event notifications*

mm_sync_events_register()

*Register or unregister for **mm-sync** event notifications*

Synopsis:

```
#include <mmsync/mmsyncclient.h>

int mm_sync_events_register( mmsync_hdl_t *hdl,
                             struct sigevent *event )
```

Arguments:

hdl

The **mm-sync** connection handle pointer.

event

The event to deliver when an **mm-sync** event is received; set to NULL to unregister.

Library:

mmsyncclient

Description:

Register or unregister for **mm-sync** event notifications. When a synchronization is active, it sends events indicating its progress. These events include the synchronization start, errors, updates, pass completions, and the synchronization finish.

Returns:

0 on success, -1 on failure.

Related Links

mm_sync_events_get() (p. 96)

*Get the next queued **mm-sync** event*

mmsync_event_queue_size_t

Data for MMSYNC_EVENT_BUFFER_TOO_SMALL *event*

Synopsis:

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

typedef struct s_mmsync_event_queue_size {
    size_t first_event;
    size_t all_events;
} mmsync_event_queue_size_t;
```

Data:

size_t first_event

The size (in bytes) of the first queued event.

size_t all_events

The total size (in bytes) of all queued events.

Library:

mmsyncclient

mmsync_event_t

General information provided for all events

Synopsis:

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

typedef struct _mmsync_event {
    mmsync_event_type_t type;
    size_t size;
    char data[0];
} mmsync_event_t;
```

Data:

mmsync_event_type_t type

The event type, as an MMSYNC_EVENT_* constant.

size_t size

The event data size.

char data

The event data.

Library:

mmsyncclient

mmsync_event_type_t

***mm-sync** event types*

Synopsis:

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

typedef enum mmsync_event_type {
    MMSYNC_EVENT_NONE = 0,
    MMSYNC_EVENT_MS_1PASSCOMPLETE,
    MMSYNC_EVENT_MS_SYNCCOMPLETE,
    MMSYNC_EVENT_MS_UPDATE,
    MMSYNC_EVENT_SHUTDOWN,
    MMSYNC_EVENT_SHUTDOWN_COMPLETED,
    MMSYNC_EVENT_MS_SYNCFIRSTFID,
    MMSYNC_EVENT_SYNC_ERROR,
    MMSYNC_EVENT_SYNCABORTED,
    MMSYNC_EVENT_SYNC_SKIPPED,
    MMSYNC_EVENT_MS_SYNC_STARTED,
    MMSYNC_EVENT_MS_2PASSCOMPLETE,
    MMSYNC_EVENT_MS_3PASSCOMPLETE,
    MMSYNC_EVENT_MS_SYNC_PENDING,
    MMSYNC_EVENT_MS_SYNC_FOLDER_STARTED,
    MMSYNC_EVENT_MS_SYNC_FOLDER_COMPLETE,
    MMSYNC_EVENT_MS_SYNC_PRIORITY_FOLDER_STARTED,
    MMSYNC_EVENT_MS_SYNC_PRIORITY_FOLDER_COMPLETE,
    MMSYNC_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE,
    MMSYNC_EVENT_MS_SYNC_FIRST_EXISTING_FID,
    MMSYNC_EVENT_BUFFER_TOO_SMALL,
    MMSYNC_EVENT_MS_SYNC_FOLDER_TRIM_COMPLETE,
    MMSYNC_EVENT_DB_RESET,
    MMSYNC_EVENT_PLAYLIST_ENTRIES_UPDATE,
} mmsync_event_type_t;
```

Data:

MMSYNC_EVENT_NONE

Indicates there are no events in the client's queue. This event is returned if the client calls *mm_sync_events_get()* when no events have occurred.

Delivered when: Never delivered.

Event data: None.

DB tables updated: None (event is not related to database updates).

MMSYNC_EVENT_MS_1PASSCOMPLETE

The files pass of synchronization is complete.

The files pass updates the **files**, **folders**, and **playlists** tables with all files (tracks and playlists) found on the device. Items that used to exist but are no longer on the device are removed. The **mediastore_metadata** table is also updated with the synchronizer used, the device's mountpoint, and the completion time of the files pass (in the *last_sync* and *syncflags* fields). No metadata has been gathered at this point.

Delivered when: The files pass completes.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: **files**, **folders**, **playlists**, and **mediastore_metadata**.

MMSYNC_EVENT_MS_SYNCCOMPLETE

The synchronization of the mediastore is complete. All tables are as accurate as possible.

Delivered when: All synchronization passes are complete for the mediastore for which the synchronization was requested.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: **files**, **folders**, **playlists**, **playlist_entries**, **audio_metadata** (if necessary), **video_metadata** (if necessary), **photo_metadata** (if necessary), **genres** (if necessary), **artists** (if necessary), **albums** (if necessary), and **mediastores**.

MMSYNC_EVENT_MS_UPDATE

Database content related to a mediastore has been updated.

Delivered when: The **mm-sync** service has written new data to the database. This event tells the client which mediastore the changes are associated with and provides the client with information on the source of the changes.

An event of this type is delivered following each change in the synchronization pass flag settings, for each operation ID associated with a specific mediastore.

Event data: The **mmsync_ms_update_data_t** structure, which contains:

- The number of added files and folders
- The synchronization pass under which any changes were made (*flags* field)
- The synchronization operation ID
- The *timestamp* field, which is the same value as the **mm-sync** timestamp assigned to the *last_sync* field of all files entries adjusted during this update

DB tables updated: Depends on which synchronization pass (if any) changed the database.

Database changed outside of synchronization (i.e., `flags == 0`): **files**.

Files pass (`flags == MMSYNC_SYNC_OPTION_PASS_FILES`): **files**, **folders**, **playlists**, and **mediastore_metadata**.

Metadata pass (`flags == MMSYNC_SYNC_OPTION_PASS_METADATA`): **files**, **folders**, **audio_metadata** (if necessary), **video_metadata** (if necessary), **photo_metadata** (if necessary), **genres** (if necessary), **artists** (if necessary), **albums** (if necessary), and **mediastore_metadata**.

Playlist pass (`flags == MMSYNC_SYNC_OPTION_PASS_PLAYLISTS`): **files**, **folders**, **playlists**, **playlist_entries**, and **mediastore_metadata**.

MMSYNC_EVENT_SHUTDOWN

Description: A client requested the **mm-sync** service to shut down. This event informs other clients about the shutdown request.

Delivered when: Immediately after receiving the request to shut down but before the shutdown process begins.

Event data: None.

DB tables updated: None (event is not related to database updates).

MMSYNC_EVENT_SHUTDOWN_COMPLETED

The **mm-sync** service has finished its shutdown process (following a shutdown request) and is no longer active. The service must be terminated and restarted to synchronize mediastores again.

Delivered when: The shutdown process is complete and playback and synchronization have stopped.

Event data: None.

DB tables updated: None (event is not related to database updates).

MMSYNC_EVENT_MS_SYNCFIRSTFID

During the files pass, the first new file that's playable by **mm-sync** was found. This event is delivered so clients can start playback as soon as possible.

Delivered when: The first new playable file has been found and placed in the library, just after the file entries marked for deletion were removed from the database. This event is delivered during full, recursive synchronizations as well as directed synchronizations. When this event is emitted, the receiver knows that all database items are valid.

Event data: The **mmsync_first_fid_data_t** structure, which contains:

- The *fid* of the first file
- The synchronization operation ID
- The *timestamp* field, which is the same value as the **mm-sync** timestamp assigned to the *last_sync* field of all file entries modified by this update

DB tables updated: **files** and **folders**.

MMSYNC_EVENT_SYNC_ERROR

An error occurred during synchronization.

Delivered when: Various synchronization errors will cause this event to be generated; the event data indicates the exact cause.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The error type
- The synchronization operation ID
- Additional information; often, the ID of the folder in which the event occurred

DB tables updated: Depends on the error type.

MMSYNC_EVENT_SYNCABORTED

The synchronization on the mediastore was aborted before completing. This happens when the device is removed, the synchronization is cancelled by the user, or there's a serious failure.

Delivered when: The synchronization is stopped on the mediastore before completing successfully.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: None (event is not related to database updates).

MMSYNC_EVENT_SYNC_SKIPPED

The synchronization wasn't started automatically on a mediastore. The user can request a manual synchronization.

Delivered when: A mediastore is inserted and a manual synchronization can be requested.

Event data: None.

DB Tables Updated: **mediastores**.

MMSYNC_EVENT_MS_SYNC_STARTED

The synchronization has begun on a mediastore.

Delivered when: The synchronization starts on a mediastore.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB Tables Updated: **mediastores**.

MMSYNC_EVENT_MS_2PASSCOMPLETE

The metadata pass of synchronization is complete.

The metadata pass updates only some metadata tables, based on the type of media files being synchronized. The **files** table is updated to show that all metadata describing the media content is now accurate. The **mediastore_metadata** table is updated to reflect the completion time of the metadata pass (in the *last_sync* and *syncflags* fields).

Delivered when: The metadata pass completes.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: **audio_metadata**, **video_metadata**, **photo_metadata**, **genres**, **artists**, **albums**, and **mediastore_metadata**.

MMSYNC_EVENT_MS_3PASSCOMPLETE

The playlist pass of synchronization is complete.

Information on device playlists is now accurate in the database. The **mediastore_metadata** table is updated to reflect the completion time of the playlist pass (in the *last_sync* and *syncflags* fields).

Delivered when: The playlist pass completes.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: **files**, **folders**, **playlists**, **playlist_entries**, and **mediastore_metadata**.

MMSYNC_EVENT_MS_SYNC_PENDING

A synchronization was requested for a mediastore but the operation was put on the pending (waiting) list because no synchronization threads were available.

Delivered when: After the **mediastores** table is updated but there are no synchronization threads available to continue synchronizing the mediastore.

Event data: The **mmsync_sync_data_t** structure, which contains:

- The error type (if applicable)
- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_EVENT_MS_SYNC_FOLDER_STARTED

A folder has started synchronization.

Delivered when: The synchronization of a folder has started. On the files pass, this event is emitted after the folder has been inserted into the database. On the metadata pass, it's emitted just before the folder contents go through the metadata pass.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The synchronization operation ID
- The current synchronization pass
- The ID of the folder in which the event occurred
- The number of files, folders, and playlists synchronized in this pass, which is 0 for all three fields
- The *timestamp* field, which is set to 0 and not used

DB tables updated: **folders**.

MMSYNC_EVENT_MS_SYNC_FOLDER_COMPLETE

All files in a folder have been synchronized and the subfolders in the folder have been enumerated.

Delivered when: The nonrecursive synchronization of a folder has completed.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The synchronization operation ID
- The current synchronization pass
- The ID of the folder in which the event occurred

The structure also contains pass-specific information, as follows:

Files pass: The number of file, folder, and playlist entries added to the database.

Metadata pass: The number of files for which the metadata was changed. The number of folders and playlists marked as changed is always 0. The *timestamp* field stores the value of the *last_sync* column in the *folders* table.

Playlist pass: The number of playlists synchronized.

DB tables updated: **folders**.

MMSYNC_EVENT_MS_SYNC_PRIORITY_FOLDER_STARTED

A priority folder has started synchronization.

Delivered when: The synchronization of a priority folder has started.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The synchronization operation ID
- The current synchronization pass
- The ID of the folder being synchronized

DB tables updated: **folders**.

MMSYNC_EVENT_MS_SYNC_PRIORITY_FOLDER_COMPLETE

A priority folder has completed synchronization.

Delivered when: The synchronization of a priority folder, which must be nonrecursive, has completed.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The synchronization operation ID
- The current synchronization pass
- The ID of the folder in which the event occurred
- The number of files, folders, and playlists within the indicated folder synchronized in this pass

DB tables updated: **folders**.

MMSYNC_EVENT_MS_SYNC_FOLDER_CONTENTS_COMPLETE

The synchronization of all subfolders in a folder is complete.

Delivered when: The recursive synchronization of a folder has completed. This event isn't emitted for a nonrecursive synchronization of a folder.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The ID of the folder in which the event occurred
- The current synchronization pass
- The number of subfolders synchronized in this pass
- The number of files and the number of playlists synchronized in this pass, which are both set to 0 and not used
- The timestamp field, which is set to 0 and not used
- The synchronization operation ID

DB tables updated: **folders**.

MMSYNC_EVENT_MS_SYNC_FIRST_EXISTING_FID

During the files pass, the first existing file that's playable by **mm-sync** was found. This event is delivered so clients can start playback as soon as possible.

Delivered when: The first playable file that's already in the library was found on the mediastore. This event is delivered during full, recursive synchronizations as well as directed synchronizations.

Event data: The **mmsync_first_fid_data_t** structure, which contains:

- The *fid* of the first file
- The synchronization operation ID
- The *timestamp* field, which indicates the time that the files pass was started

DB tables updated: None.

MMSYNC_EVENT_BUFFER_TOO_SMALL

The event buffer on the client side is too small to fetch events from **mm-sync**.

Delivered when: A client requests an event but doesn't have enough room to hold it.

Event data: The **mmsync_event_queue_size_t** structure, which contains:

- The size (in bytes) of the first queued event
- The size (in bytes) of all queued events

DB tables updated: None.

MMSYNC_EVENT_MS_SYNC_FOLDER_TRIM_COMPLETE

All database entries no longer in a folder have been deleted from the database.

Delivered when: During the files pass, if a folder is found to have been previously synchronized, this event is emitted after all the items determined to have been removed from the folder have been deleted from the database.

After this event, users know that any database items shown as being in the folder are valid.

Event data: The **mmsync_folder_sync_data_t** structure, which contains:

- The ID of the folder in which the event occurred
- The current synchronization pass, which is always 1 (for the files pass)
- The number of files, subfolders, and playlists deleted from the folder during this pass
- The timestamp field, which is the start time of the synchronization operation
- The synchronization operation ID

DB tables updated: **files**, **folders**, **playlists**, and **playlist_entries**.

MMSYNC_EVENT_DB_RESET

All file, metadata, and playlist information has been cleared from the database.

Delivered when: During synchronization, when an Apple device instructs **mm-sync** to delete all database contents.

Event data: The **mmsync_reset_sync_data_t** structure, which contains:

- The synchronization operation ID
- A timestamp value that's set to the **mm-sync** timestamp assigned to the *last_sync* fields of all file entries impacted by the reset operation

DB tables updated: **files**, **playlists**, **playlist_data**, **mediastore_metadata**, **audio_metadata**, **video_metadata**, **ipod_metadata**, **genres**, **artists**, and **albums**.

MMSYNC_EVENT_PLAYLIST_ENTRIES_UPDATE

The number of playlist entries queried for updates has reached the configured limit on how many can be queried before an event notification is sent. If any playlist entries were updated in the database, the MMSYNC_EVENT_MS_UPDATE event is also sent.

Delivered when: After the *n*th query for a playlist entry update has been issued, where *n* is defined in the **mm-sync** configuration file.

Event data: The **mmsync_pl_entries_sync_data_t** structure, which contains:

- The synchronization operation ID
- The ID of the playlist being synchronized
- The number of playlist entries queried for an update
- The number of entries that will be checked for an update
- The OID of the last playlist entry confirmed to be in the database

DB tables updated: **playlist_entries**.

Library:

mmsyncclient

mmsync_first_fid_data_t

Data for MMSYNC_EVENT_MS_SYNCFIRSTFID *and* MMSYNC_EVENT_MS_SYNC_FIRST_EXISTING_FID *events*

Synopsis:

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

typedef struct s_mmsync_first_fid_data {
    uint64_t fid;
    uint64_t timestamp;
    uint32_t operation_id;
    uint32_t count;
} mmsync_first_fid_data_t;
```

Data:

uint64_t fid

The file ID of the first media file synchronized.

uint64_t timestamp

The time when the first media file was synchronized.

uint32_t operation_id

The synchronization operation ID.

uint32_t count

The number of times that the event was sent.

Library:

mmsyncclient

mmsync_folder_sync_data_t

Data for MMSYNC_EVENT_MS_SYNC_FOLDER_* *events*

Synopsis:

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

typedef struct s_mmsync_folder_sync_data {
    uint64_t folderid;
    uint32_t pass;
    uint32_t num_files;
    uint32_t num_folders;
    uint32_t num_playlists;
    uint64_t timestamp;
    uint32_t operation_id;
    uint32_t reserved;
} mmsync_folder_sync_data_t;
```

Data:

uint64_t folderid

The ID of the folder being synchronized.

uint32_t pass

The synchronization pass (one of the **MMSYNC_SYNC_OPTION_PASS_*** flags).

uint32_t num_files

See documentation for specific event types.

uint32_t num_folders

See documentation for specific event types.

uint32_t num_playlists

See documentation for specific event types.

uint64_t timestamp

The timestamp value assigned to the *last_sync* fields of all updated database entries.

uint32_t operation_id

The synchronization operation ID.

uint32_t reserved

Reserved for future use.

Library:

mmsyncclient

mmsync_ms_update_data_t

Data for MMSYNC_EVENT_MS_UPDATE *event*

Synopsis:

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

typedef struct s_mmsync_ms_update_data {
    uint64_t added_filecount;
    uint64_t added_foldercount;
    uint32_t operation_id;
    uint32_t flags;
    uint64_t timestamp;
    uint64_t playlist_count;
    uint64_t pass_added_filecount;
    uint64_t pass_added_foldercount;
    uint64_t pass_playlist_count;
    uint64_t playlist_item_count;
} mmsync_ms_update_data_t;
```

Data:

uint64_t added_filecount

The number of files that had information added to the database.

uint64_t added_foldercount

The number of folders that had information added to the database.

uint32_t operation_id

The synchronization operation ID.

uint32_t flags

The synchronization pass (one of the MMSYNC_SYNC_OPTION_PASS_* flags).

uint64_t timestamp

The timestamp value assigned to the *last_sync* fields of all updated database entries.

uint64_t playlist_count

The number of playlists added/updated in this pass (applicable for the files and playlist pass; otherwise 0).

uint64_t pass_added_filecount

The total number of files added/updated in this pass (accumulative).

uint64_t pass_added_foldercount

The total number of folders added/updated in this pass (accumulative).

uint64_t pass_playlist_count

The total number of playlists added/updated in this pass (accumulative).

uint64_t playlist_item_count

The number of playlist items added/updated in this pass (applicable for the playlist pass; otherwise 0).

Library:

mmsyncclient

mmsync_pl_entries_sync_data_t

Data for MMSYNC_EVENT_PLAYLIST_ENTRIES_UPDATE *event*

Synopsis:

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

typedef struct s_mmsync_pl_entries_sync_data {
    uint32_t operation_id;
    uint64_t plid;
    uint64_t last_oid;
    uint32_t count;
    uint32_t total;
} mmsync_pl_entries_sync_data_t;
```

Data:

uint32_t operation_id

The synchronization operation ID.

uint64_t plid

The ID of the playlist being synchronized.

uint64_t last_oid

The ID of the last valid playlist entry in the database.

uint32_t count

The number of playlist entries updated.

uint32_t total

The total number of playlist entries to update.

Library:

mmsyncclient

mmsync_reset_sync_data_t

Data for MMSYNC_EVENT_MS_SYNC_DB_RESET *event*

Synopsis:

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

typedef struct s_mmsync_reset_sync_data {
    uint64_t timestamp;
    uint32_t operation_id;
    uint32_t reserved;
} mmsync_reset_sync_data_t;
```

Data:

uint64_t timestamp

The timestamp value assigned to the *last_sync* fields of all updated database entries.

uint32_t operation_id

The synchronization operation ID.

uint32_t reserved

Reserved for future use.

Library:

mmsyncclient

mmsync_sync_data_t

Data for many MMSYNC_EVENT_MS_* *events*

Synopsis:

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

typedef struct mmsync_sync_data {
    uint32_t operation_id;
    uint32_t error;
} mmsync_sync_data_t;
```

Data:

uint32_t operation_id

The synchronization operation ID.

uint32_t error

An MMSYNC_SYNC_ERROR_* constant, for MMSYNC_EVENT_SYNCABORTED events only.

Library:

mmsyncclient

Error information

The **mm-sync** API defines an enumeration for listing error types and a structure for holding error information.

For **mm-sync** events that indicate an error, the associated **mmsync_event_t** structure has its *type* field set to MMSYNC_EVENT_SYNC_ERROR and its *data* field refers to an **mmsync_error_t** structure. This second structure contains the error type, synchronization operation ID, and other information such as the ID of folder where the error occurred.

mmsync_sync_error_t

Data for MMSYNC_SYNC_ERROR_* errors

Synopsis:

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

typedef struct mmsync_sync_error {
    uint32_t type;
    uint32_t operation_id;
    uint32_t param;
    uint32_t reserved;
} mmsync_sync_error_t;
```

Data:

uint32_t type

The error type (an **MMSYNC_SYNC_ERROR_*** constant).

uint32_t operation_id

The synchronization operation ID.

uint32_t param

Additional information; often, the ID of the folder in which the error occurred.

uint32_t reserved

Reserved for future use.

Library:

mmsyncclient

mmsync_sync_error_type_t

mm-sync *error types*

Synopsis:

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

typedef enum mmsync_sync_error_type {
    MMSYNC_SYNC_ERROR_NONE = 0,
    MMSYNC_SYNC_ERROR_MEDIABUSY,
    MMSYNC_SYNC_ERROR_READ,
    MMSYNC_SYNC_ERROR_NETWORK,
    MMSYNC_SYNC_ERROR_UNSUPPORTED,
    MMSYNC_SYNC_ERROR_USERCANCEL,
    MMSYNC_SYNC_ERROR_NOTSPECIFIED,
    MMSYNC_SYNC_ERROR_LIB_LIMIT,
    MMSYNC_SYNC_ERROR_FOLDER_LIMIT,
    MMSYNC_SYNC_ERROR_DATABASE,
    MMSYNC_SYNC_ERROR_FOLDER_DEPTH_LIMIT,
    MMSYNC_SYNC_ERROR_DB_LIMIT,
    MMSYNC_SYNC_ERROR_FOLDER_NONMEDIA_LIMIT,
    MMSYNC_SYNC_ERROR_FOLDER_MEDIA_LIMIT,
    MMSYNC_SYNC_ERROR_MEMORY_ALLOCATION,
} mmsync_sync_error_type_t;
```

Data:

MMSYNC_SYNC_ERROR_NONE

No synchronization error.

Delivered when: Never (placeholder).

Event data: None.

DB tables updated: None.

MMSYNC_SYNC_ERROR_MEDIABUSY

The media was busy and the synchronization wasn't allowed to start on it because of concurrency rules.

Delivered when: The synchronization should have started (but didn't) on a device.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_SYNC_ERROR_READ

A read error prevented the device from being synchronized. This can be caused by a scratched disc, for example.

Delivered when: The synchronization fails because of a read error.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_SYNC_ERROR_NETWORK

A network error occurred during the synchronization. This can be caused by a metadata lookup that couldn't access the network.

Delivered when: The synchronization experiences a network error.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_SYNC_ERROR_UNSUPPORTED

The type of mediastore to synchronize isn't supported by **mm-sync**.

Delivered when: The **mm-sync** service starts the synchronization but determines the device is unsupported.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_SYNC_ERROR_USERCANCEL

The synchronization was stopped by a client request.

Delivered when: A client requests to stop the synchronization.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: **mediastores**.

MMSYNC_SYNC_ERROR_NOTSPECIFIED

A nonspecified error occurred. This error isn't classified by other error types.

Delivered when: Any time during synchronization.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: None.

MMSYNC_SYNC_ERROR_LIB_LIMIT

The files pass of synchronization reached a configuration limit; no more entries may be added to the **files** table.

Delivered when: When the maximum number of database entries has been reached during the files pass.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID
- The limit that the mediastore reached (in *param*)

DB tables updated: None.

MMSYNC_SYNC_ERROR_FOLDER_LIMIT

The files pass of synchronization reached a configuration limit; no more entries may be added to the **folders** table.

Delivered when: When the maximum number of folder items presented to **mm-sync** has reached the limit on how many folders can be scanned for synchronization.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID
- The ID of the folder in which the limit was reached (in *param*)

DB tables updated: None.

MMSYNC_SYNC_ERROR_DATABASE

The synchronization encountered a database problem.

Delivered when: A database operation fails during synchronization.

Event data: TBD.

DB tables updated: None.

MMSYNC_SYNC_ERROR_FOLDER_DEPTH_LIMIT

The **mm-sync** service skipped synchronizing a folder to avoid exceeding the configured maximum folder depth.

Delivered when: The first time **mm-sync** skips a folder because it has reached the configured maximum folder depth.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID
- The ID of the folder in which the limit was reached (in *param*)

DB tables updated: None.

MMSYNC_SYNC_ERROR_DB_LIMIT

The maximum database size has been reached; no further data may be added to the **files** and **playlist** tables.

Delivered when: When **mm-sync** notices that the maximum database size has been reached and stops the synchronization to keep the database size manageable.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: None.

MMSYNC_SYNC_ERROR_FOLDER_NONMEDIA_LIMIT

The files pass of synchronization reached the limit for nonmedia files; no entries for this folder will be added to the **files** table.

Delivered when: When the maximum number of nonmedia items in a folder has reached the configured limit.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID
- The ID of the folder in which the limit was reached (in *param*)

DB tables updated: None.

MMSYNC_SYNC_ERROR_FOLDER_MEDIA_LIMIT

The files pass of synchronization reached the limit for media files; no more entries for this folder will be added to the **files** table.

Delivered when: When the maximum number of media items in a folder has reached the configured limit.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID
- The ID of the folder in which the limit was reached (in *param*)

DB tables updated: None.

MMSYNC_SYNC_ERROR_MEMORY_ALLOCATION

The service had problems allocating memory.

Delivered when: When the service can't allocate sufficient memory during a synchronization.

Event data: The **mmsync_sync_error_t** structure, which contains:

- The synchronization operation ID

DB tables updated: None.

Library:

mmsyncclient

Index