Multimedia Detector Configuration Guide

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

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

Electronic edition published: Thursday, February 20, 2014

Table of Contents

About This Guide

The *Multimedia Detector Configuration Guide* is aimed at users of the QNX CAR Platform for Infotainment who want to understand how the `mm-detect` service discovers media devices and initiates the uploading of their metadata to persistent storage. Knowing how `mm-detect` works is essential if you want to reconfigure the service to customize its behavior.

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

To find out about:	Go to:
The role of `mm-detect` in car infotainment systems and the components it uses	*Multimedia Detector Overview* (p. 9)
Starting `mm-detect` automatically during bootup	*Running mm-detect* (p. 15)
The command-line syntax for starting `mm-detect`	*mm-detect command line* (p. 21)
The procedure used by `mm-detect` to respond to device insertions and removals	*Device tracking and media information management* (p. 23)

Typographical conventions

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

The following table summarizes our conventions:

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

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

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

Technical support

Technical assistance is available for all supported products.

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

Chapter 1
Multimedia Detector Overview

The multimedia detector service, mm-detect, keeps track of which devices are attached to a car's infotainment system and triggers the synchronization of device metadata with QDB databases.

The `mm-detect` service ensures that HMI applications can display up-to-date media information to the end user, while relieving those applications from having to constantly monitor devices and synchronize databases.

Detecting devices, retrieving their media information, and delivering that information to HMI applications involves several low-level device management and data publishing tasks. However, `mm-detect` doesn't perform most of these tasks. Instead, the service relies on the following components:

Device publishers

Inform `mm-detect` through the Persistent Publish/Subscribe (PPS) service of device attachments and detachments.

Media content detector (mcd)

Mounts filesystems of attached devices so their contents can be read by multimedia services and other programs.

mm-sync

Synchronizes the contents of QDB databases with the latest media metadata on attached devices.

CAR HMI

Monitors PPS for synchronization statuses published by `mm-detect` and queries device databases to obtain up-to-date media information for displaying to the end user.

Components used to retrieve media information

Several CAR components—device publishers, `mcd`, `mm-sync`, and the CAR HMI—work with each other and with `mm-detect` to automate the process of detecting devices and delivering up-to-date media information to the end user.

Device publishers

Device publishers are hardware-support components that detect device attachments and, in response, write PPS objects to the `/pps/qnx/mount/` directory. We refer to this fixed location as the *device listings directory*. For information on the PPS service, see the *PPS Developer's Guide*.

For each media device, the appropriate device publisher creates a dedicated PPS object to hold that device's information, which includes its mountpoint and filesystem information. For example, when the user inserts a USB stick whose device name is `usb1`, the `usblauncher` publisher creates a PPS object named `/pps/qnx/mount/usb1`. When the user removes the device, the same publisher deletes the PPS object.

When the user attaches a USB device that doesn't have any partitions but does have a single, mountable filesystem, `usblauncher` writes the device mountpoint to one PPS object whose name matches that of the device. If the USB device has partitions, the publisher writes the mountpoints of the various partition filesystems to distinct PPS objects. The names of these objects contain the partition indexes.

Media content detector (mcd)

The media content detector utility (`mcd`) enumerates and mounts devices and their partitions. The `mcd` service doesn't interact with `mm-detect`; instead, `mcd` runs as a background process and continuously monitors the `/dev/` directory for new entries, which indicate device attachments. When it notices a device attachment, `mcd` enumerates and mounts the device's partitions based on the rules encoded in a special text file (by default, `/etc/mcd.mnt`).

The mount rules indicate the mountpoints to use for specific device objects (`/dev/` entries). Device drivers create and name these objects based on partition properties such as filesystem type. For example, for an inserted USB mass storage device with distinct DOS and QNX4 partitions, the USB driver could create the device objects `/dev/umass/umass0t1` and `/dev/umass/umass0t7`. The `mcd` service could then assign mountpoints of `/fs/usb0` and `/fs/usb1` for these two partitions. For further explanation of mount rules and a sample `mcd.mnt` rules file, see the `mcd` section in the Utilities Reference.

The mcd service sends mount requests to the appropriate device drivers (e.g., the USB driver for mounting a USB stick), which manage the mountpoint paths. Each device publisher regularly polls its corresponding device driver to learn of any new mountpoints, which it then writes to the PPS objects in the device listings directory.

Multimedia synchronizer (mm-sync)

The multimedia synchronizer service, mm-sync, uploads media metadata from attached devices into QDB databases. Each device has its own QDB database that stores metadata on the device's media contents. For information on the QDB database service and the storage standard used for its databases, see the QDB Developer's Guide.

The first time a device is attached to the in-car system, mm-detect loads and initializes the device's database before invoking mm-sync to start the synchronization. This action by mm-detect is necessary because although mm-sync can write to databases, it can't load or unload them.

When a device is attached at *any* time, mm-detect instructs mm-sync to synchronize media content from the entire device, starting from the root folder and descending into all subfolders systematically. The device's filesystem is accessed from the mountpoint previously set up by mcd. The command sent to mm-sync also instructs it to upload file information and creation and playback information for media tracks, but not playlist entry information.

CAR HMI

The CAR HMI consists of many applications that provide users with a visually rich front end for interacting with their in-car system. This release contains a media app (Media Player) that reads the synchronization statuses published by mm-detect through PPS and refreshes the list of media sources shown in the HMI as needed.

You can write your own media applications to replace or run alongside Media Player. These applications can read the same PPS object that Media Player reads for synchronization status updates. The information published in this object includes the device's name and media type, and a flag indicating whether the device has been synchronized.

Your applications can extract as much of this information as they need to refresh their HMI display. By default, mm-detect stores this object in /pps/services/mm-detect/, but you can reconfigure which directory this object is stored in.

Deciding whether to run mm-detect

You can choose whether or not to run `mm-detect` in your car system. The `mm-detect` service is a convenience feature for automating the delivery of media information to client applications, but clients can also obtain this information manually.

By default, `mm-detect` is running on QNX CAR systems. The `mm-detect` process is started at boot time by the System Launch Monitor (SLM) service. You can modify the SLM configuration file (`/etc/slm-config-all.xml`) to prevent `mm-detect` from being started, as explained in the *Preventing mm-detect from running* (p. 18) section.

When to use mm-detect

You should use `mm-detect` if you want to:

- simplify client applications so they don't have to constantly scan the device listings directory to detect new devices and then manually load and synchronize the databases for those devices
- guarantee that devices have all their media information synchronized as soon as they're inserted
- guarantee that the media information for different devices is synchronized in the same order in which those devices are inserted
- limit the number of QDB databases kept in memory by automatically unloading device databases when the applications using them terminate

When not to use mm-detect

You should *not* use `mm-detect` and instead manage devices from client applications if you want to:

- synchronize some but not all media information from an attached device; this can improve performance by avoiding potentially expensive synchronizations of unneeded media content
- control the order in which media information for different devices is synchronized, to prioritize synchronizations in response to user actions
- defer synchronizing media information from a device until the client application reaches a certain state (for example, its HMI has finished loading)
- prevent synchronizing media information from certain types of devices to enforce an application policy or design

Detecting and synchronizing devices manually

To detect device insertions, a client application must constantly monitor the device listings directory (`/pps/qnx/mount/`). For any new PPS object written into this

directory, the client must read the object's `id` and `mount` attributes to obtain the device's unique ID and mountpoint. The unique ID is used to construct the database device path and the name of the database configuration object.

Before it can synchronize any media metadata, the client must write the database configuration object to the appropriate PPS directory (`/pps/qnx/qdb/config/`) to load the database that stores media information for the newly inserted device. The client can then pass the database device path and the mountpoint to the `mm-sync` service to start synchronizing the device's media metadata.

For the full details on all the steps a client must perform to manually detect and synchronize devices, see the "Synchronizing Multimedia Content" section of the *Multimedia Synchronizer Developer's Guide*.

To use Media Player when you're not running `mm-detect`, you must modify the application to manually detect devices and synchronize their media metadata.

Customizing mm-detect

The `mm-detect` program shipped with the QNX CAR Platform for Infotainment provides a basic service to synchronize all the media content from a device each time it's attached. You can do this from your client applications or write a tool to replace `mm-detect`, but you may find it most convenient to just reconfigure `mm-detect` to change how media information gets synchronized.

You can customize the behavior of `mm-detect` through command options. For example, you can name alternative schema files for defining and populating the databases that hold media metadata.

If you use a nondefault database schema, you must modify the `mm-sync` configuration file to redefine the mapping of metadata content to database fields, based on your new schema. For details on the `mm-sync` configuration file, see the *Multimedia Synchronizer Developer's Guide*.

To pass different command options to the `mm-detect` process when it's started, you must change the configuration file used by the SLM utility, which launches `mm-detect` during bootup. For instructions on specifying the `mm-detect` command line in the SLM configuration file, see the *Running mm-detect* (p. 15) section.

Chapter 2
Running mm-detect

Client applications don't run the mm-detect service at specific times to perform media tasks (like they do with other multimedia components). CAR systems use the System Launch Monitor (SLM) service to automate starting mm-detect during bootup. Applications can start mm-detect manually for recovery purposes.

Starting mm-detect with specific command options during bootup

SLM automates process management by launching processes in an order that respects their interprocess dependencies. The list of processes to launch and their properties, including their command-line arguments and interprocess dependencies, are written in a configuration file (`/etc/slm-config-all.xml`). During bootup, SLM reads this file and carries out its instructions when starting processes.

Using SLM to start `mm-detect` ensures that the system is ready to play media when the HMI finishes booting and also that `mm-detect` runs with the same command options and therefore behaves consistently from one bootup to the next. For the full explanation of how SLM works, refer to "System Launch and Monitor (SLM)" in the *System Services Reference*.

SLM is preconfigured to start `mm-detect` with certain command options but you can change which command options it passes to `mm-detect` to suit the needs of your CAR system and applications.

To change the command options passed to `mm-detect`:

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

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

   For the full list of command-line options, see "*mm-detect command line* (p. 21)".
4. Save the changes to the SLM configuration file and return to the console.
5. If you want the new configuration to take effect immediately, in the console, enter `reboot`.

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

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

Restarting mm-detect manually

An application can restart `mm-detect` with an explicit command if `mm-detect` has unexpectedly stopped and if the application can't proceed without automated media detection and synchronization.

The exact conditions that require an application to restart `mm-detect` are:

- the `mm-detect` process has terminated abnormally
- the client is required, by design or user request, to ensure that all its media information is up to date before refreshing its display
- the car infotainment system can't be rebooted to restart `mm-detect` with SLM because doing so would be too disruptive to the user

To restart `mm-detect` manually, your application must:

1. Confirm that `mm-detect` isn't already running by checking the list of active processes with the `pidin` or `ps` command.
2. Confirm that both the `qdb` and `mm-sync` processes are already running by checking the same list of active processes.

   The `mm-detect` process depends on both these other services, so if either one is *not* running, that service must be started; otherwise, `mm-detect` won't do anything when it runs.
3. Send the command line for running `mm-detect` with the desired command options to the OS, using the *system()* or *spawn()* system call.

   For details on the system calls that send commands to the OS, see the *C Library Reference*. For the full list of command-line options, see the *mm-detect command line* (p. 21) section.

The OS tries to run `mm-detect` and reports the operation outcome to `sloginfo`. If `sloginfo` shows no error, `mm-detect` is now running.

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

Preventing mm-detect from running

By default, `mm-detect` is running on CAR systems. If you decide to manually detect and synchronize media devices in client applications, you must prevent SLM from starting `mm-detect` during bootup by editing the SLM configuration file.

The version of the Media Player application shipped with the QNX CAR Platform for Infotainment won't function properly if you don't run `mm-detect`. To use Media Player without `mm-detect`, you must modify the application to manually detect device attachments and synchronize media metadata, as explained in "*Detecting and synchronizing devices manually* (p. 12)".

To prevent SLM from launching `mm-detect`:

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

   This component is the `<SLM:component>` XML object with the name `"mmdetect"`.
3. Disable the component by commenting it out (using the XML syntax of <!-- and -->) or by deleting it.
4. Save the changes to the SLM configuration file.

   In subsequent bootups of the car system, `mm-detect` won't be started.

mm-detect SLM specification

SLM uses an XML configuration file to store the list of processes to be automatically launched and their properties. In QNX CAR systems, `mm-detect` and its prerequisite and dependant programs are listed in the SLM configuration file. You can change the configuration to run `mm-detect` with different settings.

The following excerpt from the SLM configuration file shows some of the property settings for `mm-detect`:

```
<SLM:component name="mmdetect">
    <SLM:command>mm-detect</SLM:command>
    <SLM:args>-v</SLM:args>
    <SLM:depend>mmsync</SLM:depend>
    <SLM:depend>usblauncher</SLM:depend>
</SLM:component>
```

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

Command-line arguments

The `<SLM:args>` tag lists the command-line arguments. By default, only the verbosity setting (-v) is specified, but you can change the value assigned to `<SLM:args>` to include other options. The new `mm-detect` settings will take effect after you reboot the car system, causing SLM to relaunch the service. See "*Customizing mm-detect* (p. 14)" for instructions on changing the command settings for `mm-detect`.

Workflow

The `<SLM:depend>` objects list which processes must be running before `mm-detect` can be started. Based on these listed dependencies, SLM must ensure that the `mm-sync` and `usblauncher` processes are running before starting `mm-detect`.

The programs that are prerequisites to `mm-detect` have their own prerequisites. For instance, `mm-sync` requires QDB to be running so that client applications can load the media databases before `mm-sync` starts synchronizing device contents. The interprocess dependencies between `mm-detect` and the programs it uses make up a complex workflow of processes. The following is an illustration of part of this workflow:

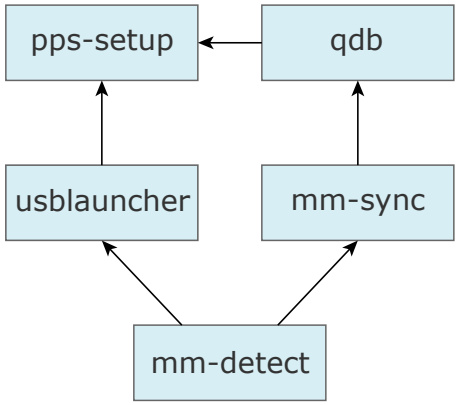

Figure 1: Workflow of mm-detect and related processes

mm-detect command line

Start the multimedia device detection service

Synopsis:

```
mm-detect [-d database_path] [-i schema_data_file]
    [-s schema_file] [-p notify_path] [-b] [-n priority] [-v]
```

Options:

-d *database_path*

Specify an overridden directory location for storing the media metadata databases.

The default is `/db/`.

-i *schema_data_file*

Name the file (with an absolute path) that defines the initial data to store in each media metadata database. This text file contains the SQL commands to populate a database when it's created, which happens when the device is inserted for the first time. Note that this option is processed only if the -s option is set to use a schema creation file.

The default is `/db/mmsync_data_car2.sql`.

-s *schema_file*

Name the file (with an absolute path) containing the SQL commands that create the initial schema of tables, indexes, and views of each media metadata database. The schema file is used only to set up the database when the device is inserted for the first time (i.e., the database didn't already exist).

The default is `/db/mmsync_car2.sql`.

-p *notify_path*

Specify an overridden directory location for storing the PPS object used to notify the car's HMI of synchronization status updates.

The default is `/pps/services/mm-detect/`.

-b

Run the `mm-detect` process in the foreground. This option is handy for debugging, because it makes `mm-detect` log messages to standard error in addition to `sloginfo`.

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

-n ***priority***

Set the priority of the `mm-detect` process. When your system is busy running many applications, the priority level can have a considerable impact on the time to see up-to-date media information after a device is inserted.

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

-v

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

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

Description:

The `mm-detect` command starts the multimedia detection service, which continuously monitors device attachments. Each time a device is attached, the service invokes `mm-sync` to synchronize the device's media metadata with the appropriate QDB database, and then publishes synchronization updates through PPS to the HMI.

Through command options, you can set new locations for storing the PPS synchronization objects, media databases, or the files containing the schemas used to create and initialize those databases. You can further control `mm-detect` by setting the command-line options for the process priority and the debugging output level.

The `mm-detect` service runs as a self-contained process that doesn't require any user input or accept any commands. It has no client utility for performing specific device-monitoring tasks on request or for adjusting any of the previously mentioned settings. To reconfigure `mm-detect`, you must change the options in its command line. In CAR systems, this command line is contained in the SLM configuration file, located at `/etc/slm-config-all.xml`.

Chapter 3
Device tracking and media information management

The mm-detect service tracks which devices are attached to the QNX CAR system, coordinates the extraction of media information from those devices, and publishes that information for client applications to read.

These tasks are performed in an automated, ongoing procedure. Understanding this procedure is essential if you wish to write a replacement utility for detecting and synchronizing devices or to customize the configuration of `mm-detect` in your car system.

The `mm-detect` process performs the following steps:

1. **Command-line parsing**

   At startup, `mm-detect` parses its command-line options, which may define:

   - the location of the PPS object used for sending synchronization status updates to the car's HMI
   - process metrics (e.g., priority level) and user and group IDs
   - the paths of the SQL files defining the schemas and initial data for the multimedia databases
   - the directory for storing the multimedia databases

2. **PPS setup**

   The `mm-detect` process creates one PPS object for storing synchronization status updates from all devices. By default, this object's full path is `/pps/services/mm-detect/status`. If the -p command option was provided to `mm-detect`, a PPS object named `status` is created in the directory named by this option's value.

   The device listings directory, which stores the PPS objects that hold device information, is a fixed location (`/pps/qnx/mount/`) that can't be changed through command options. During startup, `mm-detect` begins monitoring this directory's contents by issuing an *open()* call on the special `.all` object in this directory. By opening the `.all` object, `mm-detect` receives notification of any changes to any PPS objects in this directory. We refer to this special object as the *PPS devices information object* (see *PPS devices information object* (p. 30) for further details).

   The *open()* call is executed with the blocking flag set, meaning that the call doesn't return until there's new data from PPS.

3. **Device insertion handling**

When a media device is inserted into the car system, `mm-detect` reads the new contents of the PPS devices information object to obtain the unique ID and mountpoint of the newly inserted device. The former field is specified as the name of the device's database to the QDB service, which loads that database. The latter field is passed as an argument to a new thread that is created to synchronize the device's media metadata with the device's database.

For the full details of the actions done by `mm-detect` in response to device insertions, see *Device insertion handling* (p. 25).

After handling a device insertion, `mm-detect` issues another *open()* call to resume monitoring the device listings directory.

4. **Device removal handling**

   When a media device is removed from the car system, `mm-detect` notices the device's information is deleted from the PPS devices information object. The `mm-detect` service stops any active synchronization of the device's media contents, sends notification of the device's removal to the car's HMI through PPS, and deletes the device's database configuration file, forcing QDB to unload the database.

   For the full details of the actions done by `mm-detect` in response to device removals, see *Device removal handling* (p. 29).

   After handling a device removal, `mm-detect` issues another *open()* call to resume monitoring the device listings directory.

The `mm-detect` service runs continuously, so automated device monitoring and media synchronization will continue until the car system is shut down.

Device insertion handling

When `mm-detect` learns of a device insertion, it reads the device's information through PPS and then spawns a thread for synchronizing the device's media metadata and publishing synchronization status updates through PPS. These actions ensure that client applications can read up-to-date device status and media metadata from the databases and PPS.

The `mm-detect` process performs the following actions in response to a device insertion:

1. **PPS object parsing**

   When the user inserts a device, the appropriate device publisher detects the insertion and then writes a PPS object to the device listings directory. The content of this new PPS object is immediately added to the PPS devices information object (the `.all` object) and is returned by the `mm-detect` call to *open()*.

   This new content includes:

   - the unique ID of the device
   - the device mountpoint
   - the volume name
   - the filesystem type

2. **Mountpoint linking**

   For each newly inserted device, `mm-detect` creates a symbolic link whose target is the device's mountpoint. The link source (i.e., the referring filesystem entry) is given a path of: `/apps/mediasources/db<UID>`, where `<UID>` is the device's unique ID. Creating these links offers client media applications the convenience of accessing the filesystems of any attached device from a standard location.

3. **Database loading**

   The unique ID field is used as the name of the database's configuration object, which `mm-detect` writes to the PPS database configuration directory (`/pps/qnx/qdb/config/`). When a PPS object is added to the configuration directory, the QDB database server attempts to load the database with the same name as the object; if necessary, QDB creates that database. For more details on QDB's operation, see the QDB Developer's Guide.

   After writing the database configuration object, `mm-detect` begins monitoring changes in the PPS database status directory (`/pps/qnx/qdb/status/`). When an object with the same name as the newly inserted device is created or updated, `mm-detect` checks if that object's `Status` attribute is set to `Valid`. If so, it proceeds to the next step. Otherwise, `mm-detect` resumes monitoring the status directory to wait for the device's database to be loaded successfully.

The `mm-detect` process keeps track of which device databases are loaded. If an existing PPS object in the device listings directory is updated, for example due to a changed mountpoint, `mm-detect` recognizes from the extracted unique ID that the device's database is already loaded and so avoids unnecessarily rewriting the device's database configuration object.

4. **Media synchronization**

   Once the device's database is loaded, `mm-detect` spawns a new thread that oversees the synchronization of the device's media metadata. This thread starts synchronizing information on media files stored over the entire device and then updates the PPS synchronization status object when specific phases of synchronization complete. For details of all the actions done by `mm-detect` media synchronization threads, see *Media synchronization thread actions* (p. 26).

   Creating a dedicated thread for each synchronization ensures responsiveness, because `mm-detect` can promptly react to other device insertions or removals while still synchronizing a recently attached device.

Media synchronization thread actions

The media synchronization threads spawned by `mm-detect` handle all the interaction with `mm-sync` necessary to synchronize a device's media metadata. These threads also publish synchronization status updates through PPS, so client applications can learn when the media information in device databases is up to date.

Learning the procedure followed by the media synchronization threads teaches you how to invoke `mm-sync` from another program to ensure a device is properly synchronized. This is essential if you choose to write your own version of `mm-detect`.

When creating a thread for synchronizing a device's media metadata, `mm-detect` passes all the information it has on the newly attached device to the new thread. This information includes the device's unique ID (UID), which is also the name of the device's QDB database, and the device's mountpoint.

Each media synchronization thread performs the following actions:

1. **Connecting to `mm-sync`**

   The thread connects to the multimedia synchronizer service (`mm-sync`) by calling *mm_sync_connect()*.

2. **Registering for events**

   To register for `mm-sync` event notifications, the synchronization thread calls *mm_sync_events_register()*, specifying the system event type (a pulse) and code (the pulse type) to be delivered by the OS to indicate an `mm-sync` event has occurred.

3. **Starting media synchronization**

The thread calls *mm_sync_start()* to start synchronizing the device's media metadata. The paths of the filesystem object representing the device's QDB database (`/dev/qdb/<UID>`) and of the device's mountpoint are provided as parameters in the *mm_sync_start()* call.

The API call parameters also specify the root directory (`"/"`) as the synchronization path and set the recursive option, instructing `mm-sync` to search all directories on the device for media metadata. Also included as call parameters are the flags to synchronize the file information and media creation and runtime information, but not the flag for synchronizing playlist entry information.

4. **Sending of first status update**

   Just after starting the synchronization, the `mm-detect` thread sends a status update to the HMI through PPS. The device status is written to a special PPS object (see *PPS synchronization status object* (p. 31)).

   The status summary includes the synchronization completion flag, which is set to "false" initially to show that the device is still being synchronized. A client application within the HMI can read the status from the PPS object and learn that a device has been inserted, but its media information has not yet been updated to its database.

5. **Sending of second status update**

   The `mm-detect` thread monitors system events as it waits for the synchronization to complete. When the thread receives from the OS the type of pulse it registered for in Step *2* (p. 26), the thread calls *mm_sync_event_get()* to read the `mm-sync` data.

   If the event data indicates that `mm-sync` has completed the file synchronization pass, `mm-detect` sends another status update through PPS. This time, the synchronization completion flag is set to "true" to show that the media file information in the device's database is now up to date. A client application can then display all the device's media information to users.

   

   If the synchronization is aborted by `mm-sync`, the `mm-detect` thread receives different event data that indicates a failed synchronization operation. See the *Unexpected synchronization termination* (p. 28) subsection for details.

   ---

6. **Waiting for synchronization to complete**

   After sending the second status update, the synchronization thread continues to monitor system events until it receives another pulse indicating that an `mm-sync` event has occurred. When the event data indicates that all synchronization passes have been completed, `mm-detect` updates the device's information to reflect the successful synchronization, and then proceeds to its shutdown tasks.

7. **Deregistering for events**

To stop `mm-sync` from sending further event notifications, `mm-detect` calls *mm_sync_events_register()*, specifying `NULL` as the event type.

8. **Disconnecting from `mm-sync`**

   The `mm-detect` thread closes its connection to the multimedia synchronizer service by calling *mm_sync_disconnect()*.

Unexpected synchronization termination

If the `mm-sync` synchronization operation fails unexpectedly, the `mm-detect` media synchronization thread receives a pulse from the OS indicating that an `mm-sync` event has occurred, and then calls *mm_sync_event_get()* to examine the event data. In this case, the event data indicates an aborted synchronization, so `mm-detect` updates its information for the device to show that the synchronization failed, and then begins its shutdown activities.

If the user removes (detaches) a device while that device is still being synchronized, `mm-detect` sends a special pulse to the active synchronization thread associated with the newly removed device. This pulse contains a code that tells the synchronization thread to terminate, which it does by proceeding to its shutdown tasks.

Device removal handling

When `mm-detect` learns of a device removal, it stops any ongoing media synchronization of the device, notifies the car's HMI of the device's removal, and unloads the device's database. These actions ensure that media applications can promptly inform users of a device's removal and that system resources are quickly freed up for other tasks.

The `mm-detect` process performs the following actions in response to a device removal:

1. **Media synchronization stopping**

   When the user removes (detaches) a device, the appropriate device publisher deletes the corresponding PPS object from the device listings directory. The content of this PPS object is immediately deleted from the PPS devices information object (the `.all` object), which is constantly monitored by `mm-detect`. The `mm-detect` call to *open()* then returns a string consisting of a minus sign (`-`), which indicates an object removal, followed by the device name.

   This last field is used by `mm-detect` to retrieve the device's information, which includes a synchronization completion flag. If this flag is unset, `mm-detect` sends a terminate signal to the active synchronization thread.

2. **Mountpoint link removal**

   The `mm-detect` process deletes the symbolic link whose target is the newly removed device's mountpoint. The full path of this symbolic link is: `/apps/mediasources/db<UID>`, where `<UID>` is the device's unique ID. Deleting this link removes the device's filesystem from the view of client applications, which can no longer access the device.

3. **Synchronization status updating**

   To notify the car's HMI applications of the device's removal, `mm-detect` writes a string of the form `-db<UID>` to the PPS synchronization status object. Any application monitoring this object for updates will quickly learn of the device's removal and can update its HMI display to inform users.

4. **Database unloading**

   The `mm-detect` process deletes the device's database configuration object from the PPS database configuration directory (`/pps/qnx/mount/config/`). When the QDB database server notices the deletion of this PPS object, the server unloads the database with the same name as the object. This design supports better system performance because databases are removed from memory as soon as their corresponding devices are detached.

PPS devices information object

The PPS devices information object (`/pps/qnx/mount/.all`) contains information on all the devices currently attached to the car infotainment system.

Each attached device has its own PPS object, stored in the device listings directory. The `.all` object in this same directory concatenates the contents of all other PPS objects and therefore contains a separate section describing each device. Each of these sections begins with a line in the form: `[n]@<PPS_object_name>`. The `[n]` token means the information doesn't persist across reboots, so when the system shuts down, the device's information isn't written to permanent storage by PPS.

Following the section identifier, the device parameters are specified, one per line, in lines of the form: `key::value`. The device parameters store the hardware type, device mountpoint, volume name, database storage path, and filesystem information.

The full contents of this PPS object look like this:

```
[n]@mme
device_type::hdd
fs_type::qnx
id::mme
mount::/accounts/1000/shared/
name::Juke Box
[n]@umass0
PPS_DRIVER_ID::/pps/qnx/driver/3678286
blocks_size::512
blocks_total::15240576
partition_count::1
raw::/dev/umass0
[n]@umass0.0
PPS_DRIVER_ID::/pps/qnx/driver/3678286
PPS_RAWMOUNT_ID::/pps/qnx/mount/umass0
blocks_size::512
blocks_total::15240574
fs_type::dos (fat32)
id::4f587192-d2fe-4efb-9fec-cd35531cfa45
label::UNTITLED
mount::/fs/usb0
name::UNTITLED
partition::/dev/umass0t11
partition_order::0
plugin_name::generic
raw::/dev/umass0
read_only::false
```

The `mm-detect` process looks for the at sign (@) to mark a new section and reads the string immediately following that symbol as the name of the PPS object (and hence, the name of the QDB database) representing the attached device.

When a device is removed, the delta mode reading of the `.all` object done by `mm-detect` means that PPS sends `mm-detect` a string containing a minus sign (`-`) followed by the PPS object (and QDB database) name. At this point, the device's PPS object and the corresponding section in the `.all` object have been deleted.

PPS synchronization status object

The PPS synchronization status object (by default, `/pps/services/mm-detect/status`) contains the latest synchronization statuses of all devices currently attached to the car infotainment system.

As the synchronization of newly attached devices progresses, the `mm-detect` service updates this PPS object while client applications monitor it to learn when media metadata has been synchronized.

The first line of this PPS object is of the form: `[n]@status`.

Following this first line, the synchronization statuses of individual devices are specified, one per line, in JSON format. The fields in the synchronization statuses include but aren't limited to:

- the device's unique ID (UID)
- the filesystem type (dos, qnx4, etc.)
- the media type (`ipod`, `usb`, etc.)
- the path of the storage file for the QDB database
- a synchronization completion flag, stored as a Boolean attribute named `synched`

The full contents of this PPS object look like this:

```
[n]@status
db4f587192d2fe4efb9feccd35531cfa45:json:{"dbpath":"/dev/qdb/4f587192
d2fe4efb9feccd35531cfa45","mount":"/fs/usb0","name":"UNTITLED","fs_t
ype":"dos (fat32)","device_type":"usb","image_path":"/apps/mediasour
ces/imagecache/db4f587192d2fe4efb9feccd35531cfa45","synched":true}
dbmme:json:{"dbpath":"/dev/qdb/mme","mount":"/accounts/1000/shared/"
,"name":"Juke Box","fs_type":"qnx","device_type":"hdd","image_path":
"/apps/mediasources/imagecache/dbmme","synched":true}
```

A client application can read the `synched` field to determine whether all media information for a device is up to date, and then refresh the HMI display as appropriate. For example, when `synched` is false, the application could show an entry for the new device in a list of attached devices, but withhold the filesystem view of that device. When `synched` is true, the application can display the information for all media files on the device, because that device's database contents are known to be up to date.

Index