QNX® Aviage Multimedia Suite 1.1.0

MME Utilities Reference

For QNX® Neutrino® 6.3.x

Published under license by:

QNX Software Systems International Corporation
175 Terence Matthews Crescent
Kanata, Ontario
K2M 1W8
Canada
Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: `info@qnx.com`
Web: `http://www.qnx.com/`

Electronic edition published February 13, 2009.

Contents

About this Reference

The *MME Utilities Reference* accompanies the QNX Aviage multimedia suite, release 1.1.0. It is intended for application developers who use the suite's MultiMedia Engine (MME) to develop multimedia applications.
The table below may help you find what you need in this book:

For information about:	See:
The sound driver for iPod digital audio devices.	**deva-ctrl-ipod.so**
The USB device enumerator	**enum-usb**
The Neutrino media filesystem	**io-fs-media**
The interface to the iPod authetication chip	**iofs-i2c-ipod.so**
The iPod device driver for **io-fs**	**iofs-ipod.so**
The PFS/MTP device driver for **io-fs**	**iofs-pfs.so**
The serial transport mechanism for connecting to iPods	**iofs-ser-ipod.so**
The USB transport mechanism for connecting to iPods	**iofs-usb-ipod.so**
The Neutrino multimedia playback controller	**io-media-generic**
The Media content detector utility (MCD)	**mcd**
The Multimedia Engine (MME) resource manager	**mme-generic**
The MME browse client	**mmebrowse**
The MME commandline client	**mmecli**
The MME explorer interface commandline client	**mmexplore**

Other MME documentation available to application developers includes:

Book	Description
Introduction to the MME	MME Architecture, Quickstart Guide, and FAQs.
MME Developer's Guide	How to use the MME to program client applications.
MME API Library Reference	MME API functions, data structures, enumerated types, and events.
MME Configuration Guide	How to configure the MME.
MME Technotes	MME technical notes.
QDB Developer's Guide	QDB database engine programming guide and API library reference.

Note that the MME is a component of the QNX Aviage multimedia core package, which is available in the QNX Aviage multimedia suite of products. The MME is the

main component of this core package. It is used for configuration and control of your multimedia applications.

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	`something you type`
Keyboard keys	Enter
Program output	`login:`
Programming constants	NULL
Programming data types	`unsigned short`
Programming literals	`0xFF`, `"message string"`
Variable names	*stdin*
User-interface components	**Cancel**

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 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 options

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

deva-ctrl-ipod.so

Sound driver for iPod digital audio devices

You must be **root** to start this driver.

Syntax:

Direct invocation (also causes a new **io-audio** process to start):

```
io-audio -dipod busno=bus,devno=device,cap_name=name,ipod_mount=path &
```

Runs on:

ARM, PowerPC, SH, x86

Options:

busno=*bus*	The USB bus number. Specify if more than one USB audio device will be plugged in.
devno=*device*	The USB device number. Specify if more than one USB audio device will be plugged in.
cap_name=*name*	Create a symbolic link to the capture device.
ipod_mount=*path*	The path to the iPod driver mountpoint.

Description:

The **deva-ctrl-ipod.so** shared object is a DLL for the **io-audio** manager. It uses the API described in the *Audio Developer's Guide*.

While **deva-ctrl-ipod.so** is running, you can use applications with sound (e.g. **mmplay**) and those that control the sound-system (e.g. **mixer**).

When you start **io-audio** with **deva-ctrl-ipod.so**, you need to specify the path to the iPod driver mountpoint: **ipod_mount**. If you will have more than one USB audio device plugged into the system, you must also specify the USB bus number: **busno**, and the USB device number: **devno**. It is good practice to always specify these three options. You can also specify the symbolic link to the correct capture device to bring things together.

For example, assuming that an iPod is plugged in (bus number 0 and device number 1) and that you mount the iPod **io-fs** driver at **/fs/ipod0**, you could start the driver as follows:

```
io-audio -dipod busno=0,devno=1, \
    cap_name=ipod-0-1,ipod_mount=/fs/ipod0 &
```

Errors:

When an error occurs, **deva-ctrl-ipod.so** sends a description of the error to the system logger (see **slogger**).

See also:

io-audio, **mixer**, *Audio Developer's Guide*, Working with iPods in the *MME Developer's Guide*

Syntax:

enum-usb [*options***]**

enum-usb [*options***,]validate**

Runs on:

ARM, PowerPC, SH, x86

Options:

Options for **enum-usb** are separated by commas, with no whitespaces. They include:

cfg_file_path=*pathname*

pathname is the path to the **enum-usb** configuration file. Default is **/etc/enum-usb.conf**.

iface_tbl_size=*size*

size is the maximum number of interfaces **enum-usb** will manage at the same time. These interfaces can be on any number of devices. Default is 50 interfaces.

USB_mgr_pathname=*pathname*

pathname is the path to the USB resource manager. Default is **/dev/io-usb/io-usb**.

validate Validate the configuration file. If this option is used, **enum-usb** will parse the configuration file, report any errors it encounters in the file, then exit.

verbose=*level* Turn on verbose mode and output to **stdout** at the verbosity level specified by *level*. When using **enum-usb** in combination with **enum-devices**, you must also set the verbose options of **enum-devices** for verbose statements to be available at the console.

Description:

This description of **enum-usb** includes the following sections;

- Overview
- USB device information
- Behavior when enumerating a single USB device
- The **enum-usb** configuration file
- **enum-devices**

Overview

enum-usb is a bus-enumerator program spawned by **enum-devices** to handle USB-specific aspects of device enumeration.

enum-usb scans the USB bus and writes a line describing each device found to standard output (**stdout**). The device enumerator manager **enum-devices** reads in the information from this stream and launch the drivers required to manage the devices via the **enum-devices** configuration files.

When a USB device is removed from the system, **enum-usb** reports this removal to **enum-devices** so that it can remove the drivers it launched for that device.

USB device information

enum-usb provides **enum-devices** the following information about interfaces on a USB device:

- *busno* — the USB bus number
- *cfg* — the USB configuration the device is using
- *class* — the device or interface class
- *devno* — the USB device number
- *did* — the device ID
- *iface* — the USB interface number
- *num_iface* — the number of interfaces available for the selected USB device configuration
- *proto* — the device or interface protocol
- *rev* — the device version
- *subclass* — the device or interface subclass
- *vid* — the vendor ID

See the **enum-usb** configuration file's **Set** for additional information that **enum-usb** can provide to **enum-devices**.

Microsoft descriptors

If the device supports Microsoft descriptors (for example a PFS device) **enum-usb** provides the following additional information:

- *mscomp* — Microsoft compatible ID
- *msven* — Microsoft vendor ID
- *mssubcomp* — Microsoft subcompatible ID

Behavior when enumerating a single USB device

When **enum-usb** enumerates a single USB device, it *may* report more than one device to **enum-devices**. Behavior is defined by the USB device's *device class*, as follows:

class is other than zero (!=0)

enum-usb considers that any interfaces present on the device cannot be used independantly of the other interfaces on the device.

It reports one useable interface for this device, and expects that **enum-devices** will launch a single driver to manage the device and its interfaces.

class is vendor specific (0xFF)

enum-usb considers that the device will be handled by a single driver that knows how to handle this vendor specific device. It reports one useable interface for this device.

class is interface specific (=0)

enum-usb reports an event for every interface present for the current device configuration.

Each interface of this USB device can be managed independantly of the other interfaces on the device and, therefore, **enum-devices** can be configured to launch a unique driver for every USB interface reported for the USB device.

The enum-usb configuration file

The **enum-usb** configuration file allows you to identify USB devices by vendor ID, device ID, and serial number in order to control how the device is enumerated by **enum-usb**. The default path and name of the configuration file is **/etc/enum-usb.conf**. You can use the **-c** at startup to specify another the configuration.

The configuration file uses the following options:

- **Device** — start of a device configuration definition; see **Device** below.
- **Ignore** — instruct **enum-usb** to ignore the USB device; see **Ignore** below.
- **Config** — specify the USB device configuration; see **Config** below.
- **Set** — specify a special tag to include in USB device reports to **enum-devices**; see **Set** below.

Each option must be on its own, separate line. Commented lines begin with a number sign ("#") and are ignored by **enum-usb**. Blank spaces are ignored.

Example `enum-usb.conf` configuration file

```
#Specify the configuration to use for IPOD devices
Device vid=05AC,did=12*
Config class=03
Set usr_spec_id=AppleIpod

#Do not enumerate this device
Device vid=13FE,did=1D00
Ignore
```

Device

The **`enum-usb`** configuration file **`Device`** option identifies the start of a device configuration definition. All options that follow a **`Device`** option in the configuration file are applied to that device, up to a new instance of the **`Device`** option.

The **`Device`** option identifies a USB device by its:

- device ID (*did*)
- vendor ID (*vid*)
- serial number (*ser*)

The statement for the **`Device`** option takes the form:

`Device [vid=`*v***`],[did=`***d***`],[ser=`***s***`]`**

The rules for composing the **`Device`** option statement are:

- In the model above, *vvvv*, *dddd* and *s* represent the hexadecimal values to match, and *s* represents the serial number string.
- The option statement must contain a least one parameter.
- Parameters can be in any order, separated by commas.
- If the device will be ignored, do *not* specify the serial ID (*ser*) parameter; see **`Ignore`** below.
- The option statement supports wildcard matching (see *fnmatch()*), so you can match a range of devices. For example, **`did=05ac,vid=12*`**, would match a specific vendor and a range of devices, where 12* matches 1200 thru 12FF.
- If matching strings do not contain a wildcard, they must specify the vendor ID (*vid*) and the device ID (*did*) as four numerals, with leading zeros if necessary. The serial number is a string with a device specific format and length.

Ignore

The **Ignore** option instructs **enum-usb** to ignore the device and to *not* enumerate it.

This option is useful if you have a USB device for which a driver registers for the insertion event for the device itself and does not need to be launched by the enumerator. Instructing **enum-usb** to ignore this type of device, eliminates any possible conflict between the enumerator and the driver, which might both attempt to attach to the device at the same time.

If you want **enum-usb** to **Ignore** a device, the **Device** option identifying it should specify only the device ID and the vendor ID.

enum-usb ignores serial numbers defined in the **Device** option of ignored devices, because to obtain a device's serial number it must "attach" to the device, and attaching to an ignored device may create a conflict with a driver attempting to get exclusive access to the device.

Config

The **Config** option allows the user to specify which USB device configuration is used when the device to be enumerated supports multiple configurations.

If the **Config** option is not used, the **enum-usb** uses the first device configuration selected. If **Config** option is used but the specified configuration is not present on the device, **enum-usb** logs an error and does not enumerate the device.

When **enum-usb** encounters the specified device, it sets the USB device's configuration to either the number specified (**Config num=***x*) by the **Config** option, or to the first configuration with an interface that matches the class and subclass combination specified by the **Config** option.

Multiple configuration selections

If a device supports multiple configurations or interfaces from generation to generation, in your **enum-usb** configuration file you can specify multiple **Config** option lines for each Device, in priority order. **enum-usb** uses the first **Config** option statement that matches the device for which it is listed.

Composing Config option statements

Statements for the **Config** option take the form:

Config [class=*xx***],[subclass=***xx***]**

or:

Config num=*x*

The rules for composing **Config** option statements are:

- In the model above, *xx* represents the hexadecimal values for the *class* and *subclass* to match, and *x* represents the configuration decimal value to match.

- The option statement must contain a least one parameter.
- The option statement can specify any of the following:
  - only the *class* parameter
  - only the *subclass* parameter
  - both the *class* and the *class* parameter
- The matching strings for *class* and *subclass* must be two numerals, with a leading zero if necessary.

- When you specifying the configuration number, you are specifying the value reference number of the configuration, rather than its index. For example, if a device has two configurations, they may not be referred to as configuration one and configuration two. To determine the configuration value of a device configuration, refer to the output of the USB utility (**`usb -vv`**).
- QNX recommends that you use the *class* and *subclass* parameters wherever possible, rather than a configuration number (*num*). For some devices, the environment in which they are used may affect which configurations they report. This device characteristic means that in different environments, the same configuration number may represent different configurations.
- Use the configuration number parameter for the **`Config`** option *only* when you are certain that the configuration numbers you use will do not change — when you are certain that the numbers will always represent the same configurations, in all circumstances.

`Set`

The **`Set`** option allows you to specify a custom string to be passed to **`enum-devices`**. This string can be:

- used to help **`enum-devices`** match a device to a specific **`enum-usb`**configuration
- applied to an **`enum-devices`** configuration; for example, as an argument applied for launching a driver

There are currently two defined custom tags that **`enum-usb`** can pass to **`enum-devices`**:

- *user_spec_id* — user-specified identifier
- *inc_user_spec_id* — incrementing user-specified identifier

Composing `Set` option statements

`Set` statements in the **`enum-usb`** configuration file must be specified as follows:

- If the **`Device`** section does not contain a **`Config`** statement, **`Set`** statments can be specified after the **`Device`** statement.
- If configuration file includes **`Config`** statements, **`Set`** statements must be placed immediately following the **`Config`** statements to which they apply.

For example:

```
Device vid=05AC,did=12*

    Config class=01

        Set usr_spec_id=AppleIPODwithDigitalAudio

    Config class=08

        Set usr_spec_id=AppleIPODUMASSmode
```

user_spec_id

The **`Set`** *user_spec_id* option is useful for associating multiple devices, or a range of devices, with a single **`enum-devices`** configuration. It can be:

- used to confirm matching criteria, and thus launch the same driver for all the associated devices
- passed as an argument to the matched configuration, and thus represent a common system component comprised of several launched drivers

The statment for the **`Set`** *user_spec_id* option takes the form:

`Set usr_spec_id=`*x*

The rules for composing the **`Set`** *user_spec_id* option statement are:

- In the model above, *x* represents a string with no white spaces.
- The string length is limited to 40 characters.

inc_user_spec_id

The **`Set`** *inc_user_spec_id* option is useful for associating multiple devices, or a range of devices, while providing a device-unique incrementing suffix appended to the user supplied string. Because **`enum-devices`** configuration files do not support wildcards, this option can be used only as an argument passed to the matched configuration.

The suffix appended to the user-supplied string by this option:

- is taken pulled from a shared pool

- is reserved for the name provided
- is released when the device is removed
- is reusable after release
- starts incrementing from zero

For example, behavior with the sample configuration **inc_usr_spec_id=/fs/ipod** is as follows:

- When the first iPod is connected to the host system, all insertion events related to that device include the tag **inc_usr_spec_id=/fs/ipod0**.
- When a second iPod is connected to the host system, all insertion events related to that device include the tag **inc_usr_spec_id=/fs/ipod1**, and so on for all further iPods connected to the system.
- If the second iPod device is removed, the **1** suffix becomes free, and will be reused for the next iPod inserted.

- If you want the device suffix to be incremented from zero for a device entry in your **enum-usb** configuration file (**enum-usb.conf**) instead of from a shared pool, you must use a unique *inc_ usr_spec_id* for that device entry in the configuration file.
- If the *inc_usr_spec_id* value is a pathname, the basename is used to reserve the suffix: **inc_usr_spec_id=/fs/ipod**, for example, will share the suffix pool with *inc_usr_spec_id=ipod*.

The statment for the **Set** *inc_user_spec_id* option takes the form:

Set inc_usr_spec_id=*x*

The rules for composing the **Set** *inc_user_spec_id* option statement are:

- In the model above, *x* represents a string with no white spaces.
- The string length is limited to 1024 characters, including the appended suffix.

enum-devices

This section provides basic information useful for using **enum-devices** with **enum-usb**. For a complete description of **enum-devices** configuration file format, see **enum-devices** in the *Neutrino Utilities Reference*.

enum-devices matching rules

Matching behavior for **enum-devices** is based on the “number of matches” in the **enum-devices** configuration file device statement. A secondary device statement may be created by placing a period (“.”) in front of the match field.

This method for matching devices with entries in the **enum-devices** configuration file can sometimes result in "ambiguities": a specified device that matches more than one entry in the **enum-devices** configuration file, rendering it impossible for the enumerator to select the preferred driver for the device.

For example, with a device fully specified as:

```
vend=1922,dev=1234,busno=0,devno=1,
rev=100,msven=fe,mscomp=MTP,mssubcomp=0
```

and with the following in the **enum-devices** configuration file:

```
device(usb, vend=1922,dev=1234)

device(usb, class=08,proto=00)
```

the specified device matches both the entries in the configuration file.

This ambiguity can be removed by either extending the match criteria for the preferred driver:

```
device(usb, vend=1922,dev=1234,class=08)
```

or by making one of the matching fields *secondary*; that is, by giving it a lower precedence that the other fields:

```
device(usb, .class=08,proto=00)
```

A "catch all" (an action that matches any device) is produced by simply entering the bus type for the device. For example:

```
device(usb)
    echo("No match found for device ven=$(ven),
    dev=$(dev), class=$(class), busno=$(busno),
    devno=$(devno), cfg=$(cfg), iface=$(iface),
    msven=$(msven), mscomp=$(mscomp),
    mssubcomp=$(mssubcomp)" )
```

File precedence

The order of precedence for **enum-devices** configuration files is determined by one or both of the following criteria:

- the order in which configuration files are passed via the commandline **-c** option
- file precedence order in the directory

If files are added to an enumeration directory after the enumeration utility has been started, the enumerator picks up these files and assigns them a higher precedence than currently existing files.

Given the complexity associated with adding and removing files and directories during processing, QNX recommends that you:

- use a static directory and file configuration
- modify a single file in a higher precedence directory to override the behavior of existing matches

Structure of the `enum-devices` configuration file

The **enum-devices** configuration file is typically takes the following format:

```
/enum
    common
    /devices
        default
        net
        mass
        char
    /include
        usb-class
    /overrides
        mass
```

enum-devices launches **enum-usb** when it starts. To enable **enum-devices** to launch **enum-usb**, you must add **enum-usb** to the common file accesed by both **enum-devices** and **enum-usb**, as follows:

```
all
    config(include)
    config(overrides)
    config(devices)
    enumerator(usb)
```

Examples:

Below is a sample **enum-usb** command line start up that specifies the path to the configuration file and sets the interface maximum:

```
# enum-usb cfg_filepath=/etc/enum-usb-custom.conf,iface_tbl_size=32
```

enum-usb.conf

In this example implementation we want **enum-devices** to launch a driver for an iPod, which has several different device IDs. Instead of a using a **enum-devices** configuration containing an entry for every *vid* and *did* combination, we pass up a user specified ID which represents the set of iPod devices.

```
#Specify the configuration to use for IPOD devices
Device vid=05AC,did=12*
        Config class=03
        Set usr_spec_id=AppleIpod
```

enum-devices configuration file

```
# If the class is AUDIO and the subclass is STREAMING,
# and the AppleIpod string is present,
# launch io-audio.
```

```
device(usb, class=01, subclass=01, usr_spec_id=AppleIpod)
    driver(io-audio "-dipod busno=$(busno),devno=$(devno),
    cap_name=ipod-$(busno)-$(devno),
    ipod_mount=/fs/ipod-$(busno)-$(devno)")
```

See also:

enum-devices, **io-fs-media**, **iofs-ipod.so**, **iofs-pfs.so**, **iofs-ser-ipod.so**, **iofs-usb-ipod.so**, **mme**

Syntax:

io-fs-media *options*

Runs on:

ARM, PowerPC, SH, x86

Options:

-a Always start all the built-in drivers.

-b Don't run in the background (used for debugging).

-c*TYPE=num* Set a cache size. The type of cache is specified by *TYPE*, and can be one of:

- **attrs** — number of attributes (default: 32)
- **bundles** — number of 64 KB bundles (default: 2)
- **files** — maximum number of files per filesystem (default: 1 MB)
- **maps** — number of extent mappings (default: 512)
- **mcache** — maximum size of the meta cache (default: 16 KB)
- **meta** — size of the metadata cache (default: 16 KB)
- **pages** — number of 4 KB clusters (default: 256)
- **throngs** — number of 1 MB throngs (default: 0)
- **wads** — number of 256 KB wads (default: 0)

-d *driver*,*options* Start only the specified driver with the specified options. The available drivers are (see below for more information about these drivers):

- **ipod** (**iofs-ipod.so**)
- **tmp** (built into **io-fs-media**)
- **pfs** (**iofs-pfs.so**)

Because **io-fs-media** is single-threaded, you should start each driver in its own process.

-o *fsys*[=*blkpat*],[+*options*]

Override automount options for *fsys*. The "+" appends options.

-o cd[=*device*],use,nojoliet,noudf

CD filesystem options

-o lt[=*blkpat*],use,format[=*maxfile:maxmap*],sync=sec,notrans
LT filesystem options

-s Cause **rename**, **unlink**, **chmod**, **chown** and **chmod** to be synchronous.

The **-s** option applies to writable filesystems; the media filesystems supported by **io-fs-media** are read-only, and don't support this option.

-t Print the built-in tables to **stdout**, then exit.

-v Add the verbose option to all mounted filesystems.

Description:

The **io-fs-media** filesystem makes disparate media devices and filesystems appear as POSIX-compliant filesystems under QNX Neutrino. It is based on the **io-fs** filesystem framework, and provides some extensions specific to media devices. This variant of **io-fs** is designed specifically for use with the MME. For more information about the architecture of **io-fs**, see the *Introduction to the MME*.

The **io-fs-media** supports these media devices and filesystems:

- RAM filesystem (**tmp**) — a temporary POSIX filesystem in RAM. This functionality is statically linked to the **io-fs-media** executable. For information about starting and configuring this module, see "Starting the RAM Filesystem" below.
- iPod filesystem — filesystem support for accessing iPod devices, including media-specific extensions. This functionality is provided in a separate **iofs-ipod.so** library, loaded at runtime. Device driver support is provided separately by a serial or USB device driver.

  For information about starting and configuring the iPod module, see **iofs-ipod.so**.
- PFS filesystem — device-driver and filesystem support for accessing Microsoft PlaysForSure (PFS) devices that use the Media Transport Protocol (MTP). This functionality is provided in a separate **iofs-pfs.so** library, loaded at run-time. Additional libraries supporting DRM may also be used. The USB driver is required.

  For more information about starting and configuring the PFS module, see **iofs-pfs.so**. For information about configuring the module to access DRM-protected content, see "Enabling Digital Rights Management (DRM)".

Other media device support options for **io-fs-media** may be available; contact your QNX sales representative for information.

The io-fs configuration files

io-fs uses the following configuration files, located at **io-fs/lib/config/etc/**:

- **iofs.fsd** — driver startup rules
- **iofs.fsf** — filename rules
- **iofs.fsm** — auto-mounting rules table

iofs.fsd

The **iofs-fsd** configuration file sets the driver startup rules. To use a rule, simply uncomment the line for the drive.

- Use the **-d** command line option to have force start a driver.
- The **mount**= option is only available for drivers that can only mount a single known file system, such as “media”, “tmp” or “et”.
- Block devices can mount many filesystems; these are configured in the **iofs.fsm** table.

Below is a sample **iofs.fsd** configuration file.

```
# DRIVER       OPTIONS
tmp           mount=/fs/tmpfs
#file          dir=/dev:/var/images,safe
#ata
umass

#mtest         mount=/fs/mtest%#
ipod          device=/dev/ser1,mount=/fs/ipod%#
pfs           mount=/fs/pfs%#

#sim_nor       object=/nor,size=32m,mount=/fs/etfs%#
#sim_nand512   object=/nand512,size=512k,mount=/fs/etfs%#
#sim_nand2048  object=/nand2048,size=64m,mount=/fs/etfs%#
```

iofs.fsf

The **iofs-fsf** configuration file sets the filename rules used by **io-fs**.

Below is a sample **iofs.fsf** configuration file.

```
# Filename rules
# PATTERN     FLAGS
*.aac         SN P(64K)   G(64K)
*.flac/*.aac
*.mp3/*.aac
*.ogg/*.aac
*.wav/*.aac
```

```
*.wma/*.aac

*.mpg          SN P(256K)  G(64K 256K,256K 2M,1M)
*.avi/*.mpg
*.mov/*.mpg
*.mpeg/*.mpg
*.wmv/*.mpg

*.o            SN P(64K)   G(16K  16K,64K)
*.lib          SN P(64K)   G(256K)

# Next group for testing
*.non          SN P(0K)    G(0K)
*.pag          SN P(4K)    G(4K)
*.bun          SN P(64K)   G(64K)
*.wad          SN P(256K)  G(256K)
*.thr          SN P(1M)    G(1M)
*.xxx          E

!              S  P(64K)   G(16K 16K,64K)
*                          G(4K  16K,8K 64K,32K)
```

Filename rule nomenclature

The filename rules use the following nomenclature:

- E — Encrypt
- N — No re-use
- R — Random
- S — Sequential
- P(n) — Pre-read
- G(n, n,n n,n) — Growth

iofs.fsm

The **iofs-fsf** configuration file contains a table that defines the rules used by **io-fs** to auto-mount devices.

Below is a description of the contents of the **io-fs** mount rule table in the **iofs.fsm** configuration file. For more information about how **io-fs** uses the rules in the table see the comments following the table.

Pattern	Filesystem	Mountpoint	Options
dos-*	dos	**/fs/dos%#**	
nt-*	nt	**/fs/ntfs%#**	
lt-*	lt	**/fs/ltfs%#**	if=unknown,format,endif
qnx4-*	qnx4	**/fs/qnx%#**	
ext-*	ext2	**/fs/ext%#**	
cd	cd	**/fs/cda%#**	
usb-*-disk	pc	**volmgr**	if=invalid,format,endif,if=empty,add=dos:full,endif
flop-*-disk	dos	**/fs/dosflop%#**	
disk	efi	**volmgr**	
disk	pc	**volmgr**	
disk	dos	**/fs/dos%#**	if=invalid,abandon,endif
disk	qnx4	**/fs/qnx%#**	if=invalid,abandon,endif
disk	lt	**/fs/ltfs%#**	if=invalid,abandon,endif

Comments

The fields defining how **io-fs** auto-mounts a device are described below.

Pattern

The *Pattern* field sets a pattern as defined by the **fnmatch** function, which is applied to a block device containing a file system. The block device has a well-defined naming convention, as follows:

bus-busdevno-class-unit-type.fstype-partslot

Where:

- bus — the name of the bus the controller is on: “pci”, “usb”, etc.
- busdevno — a number relative to the bus (hexadecimal).
- class — the name of the controller: “ata”, “scsi”, “umass”, etc.
- unit — a unit number on the controller (decimal); for example, 0 primary master, 1 primary slave, 2 secondary master, etc.
- type — the type for the filesystem, if it is known: “disk”, “cd”, “tmp”, “flash”, etc.
- fstype — the filesystem type for the partition: “lt”, “dos”, “qnx4”, etc.
- partslot — the volume-specific slot number (decimal).

Filesystem name

The *Filesystem name* field sets the name of the filesystem to start: “lt”, “dos”, “qnx4”, “nt”, “ext2”, “cd”, etc.; or, if the mountpoint is NULL, the name of the volume manager to use; for example, “pc” for a standard Microsoft partition table.

Mountpoint

The *Mountpoint* field sets the location where **io-fs** mounts the filesystem in the pathname space.

The percent sign “%” is a special macro character with the following values defined:

- **%%** — %
- **%0** — bus, **%1** — busedevno, **%2** — class, **%3** — unit, **%4** — type, **%5** — fstype, **%6** — partslot
- **%#** — minor number of a filesystem: 0, 1, 2, etc.; each filesystem is independant, starting at 0 (zero)
- **%@** — minor number common across all filesystems

Options

The *Options* field sets filesystem-specific or volume manager-specific options. Common options include:

- **if={cond},cmd1,cmd2,else{=cond},cmd3,cmd4,endif,cmd5**
- abandon — exit without mounting
- Conditions **{cond}** or **!{cond}**:
  - unknown — the partition is not identifiable by the filesystem
  - bad — the partition is identfied, but is not usable
  - invalid — unknown or bad
  - empty — no partitions are defined in the volume

Starting the RAM Filesystem

The RAM/TMP filesystem creates a temporary POSIX filesytem in system RAM that exists for the duration of the **io-fs-media** process.

The TMP filesystem has these additional command line options:

use	Print out a usage message and then exit.
mount=*path*	The path to mount the RAM filesystem at; if not specified, the filesystem is mounted at **/fs/tmpfs**.
minsize=*megs*	The minimum filesystem size, in Mb.

maxsize=*megs* The maximum filesystem size, in Mb.

noglob Specify that the filesystem shouldn't be located in global memory.

The **noglob** option is for ARM9 targets only.

Examples:

The examples below show some **io-fs** start up options:

io-fs -b Do not put into the back ground so you can kill it with a keyboard break. Great for debugging.

io-fs -b -olt,format

Always format auto-mounted **ltfs** partitions.

See also:

iofs-ipod.so, **iofs-pfs.so**, **mme**

iofs-i2c-ipod.so

The interface to the iPod authentication chip

Syntax:

io-fs-media -dipod,acp=i2c:*options*

Runs on:

ARM, PowerPC, SH, x86

Options:

Options for **iofs-i2c-ipod.so** are separated by colons (:). These options are:

addr=*address*
: The address, in hexadecimal, to the memory location to connect to on the authentication chip. This address is usually from 0x10 to 0x17, inclusive. The default is 0x10.

path=*chippath*
: The path to the authentication chip. If this option is not specified, **iofs-i2c-ipod.so** uses the default: **path=/dev/i2c0**.

speed=*connectionspeed*
: The i2c connection speed to the authentication chip. If this option is not specified, **iofs-i2c-ipod.so** does not change the configured speed.

use
: Print out a usage message to **stdout** and exit.

Description:

The utility **iofs-i2c-ipod.so** provides the interface to the Apple authentication chip.

If instructed to do so, **iofs-ipod** can load another authentication interface. However, only **iofs-i2c-ipod.so** is currently supplied.

See also:

io-fs-media, **iofs-ipod.so**, **iofs-pfs.so**, **iofs-ser-ipod.so**, **iofs-usb-ipod.so**, **mme**

iofs-ipod.so

iPod driver for **io-fs**

Syntax:

io-fs-media -dipod,*options*

Runs on:

ARM, PowerPC, SH, x86

Options:

The **iofs-ipod** *options*, in addition to the standard **io-fs** options, include:

acp=*interface*
: The authentication chip interface. See "Authentication chip interface" below.

darates=[**+**]*rate*
: Digital audio sampling rates to support. Required for digital audio support; see "Adding digital sampling rates" below.

eq=**on | off**
: Disable the equalizer. The default value is **on**.

  To disable the equalizer, start the iPod driver with the **eq=off** option. For example:

```
# io-fs-media -dipod,transport=ser:dev=/dev/serfpga3,eq=off
```

fileburst=*number*
: The number of files to discover from the iPod at a time. By default, this value is **16**. Increasing the size of the fileburst increases the speed at which files are discovered, but requires more system resources.

fnames=**short | long**
: The type of filename **io-fs-media** should use for files on iPod devices. The default is "long", which is *not* recommended for iPod devices. The behavior and implications of the different **fnames** options are as follows:

  - long — *Not recommended*. The filenames used by **io-fs-media** closely match the iPod titles.
  - short — *Recommended*. The filenames used by **io-fs-media** are synthetic filenames that represent the iPod filenames.

  See "About filenames" below.

lang=*lang_code*
: Set the iPod language. The default language is English (en_EN).

logfile=*filepath*
: Print the log to the specified file instead of to **stout**.

mount=*path* The path to mount the iPod at. By default, the device is mounted at **/fs/ipod***num*, where *num* starts at **0** and increments with each additional device mounted.

playback Show entries representing the titles in the iPod playback engine from **/fs/ipod***n***/.FS_info/playback**. You can use these entries to access the information for the files the entries represent in the iPod's playback engine.

polltimeout=*timeout*

The timeout for polling, in milliseconds. By default, this value is **2000**.

pref=*preferences* Set preferences on iPods and iPhones at startup. See "Preferences" below.

restore=*settings* Instruct the iPod to restore or keep the specified setting when it is removed from the system. See "Restoring settings" below.

rtimeout=*time* The serial device command read timeout, in milliseconds. By default, this value is **5000**.

sndchk=**on | off | ipod**

The sound check setting. Supported values are:

- **on** — (default) perform a sound check
- **off** — do not perform a sound check
- **ipod** — let the iPod decide

splash=[*greyscale_icon_path*]:[*color_icon_path]:[big_color_icon_path*]
Splash screen images to be displayed on the iPod device. Use colons to separate the different image paths. See "Splash screens" below.

storage Turns on HD radio tagging support.

transport=*type*:*options*

Specifies the transport mechanism. Loads **iofs-***type***-ipod.so** and checks if it supports the transport interface. Only **ser** and **usb** are currently supported. See **iofs-ser-ipod.so** and **iofs-usb-ipod.so** in this *Reference*.

use Print out a usage message to **stdout** and exit.

verbose=*number* Enable verbosity levels. Higher verbosity levels provide increasing levels of detail for device transactions.

Description:

This module provides a media filesystem interface for iPod devices. An iPod can connect via its 30-pin Omni connector to a RS232 serial UART connection on the Neutrino system.

During playback, iPods usually deliver events every 500 milliseconds. This interval is not configurable.

Enabling MME support for iPods

To enable support for iPod devices with the MME, the contents of the MME's **slots** table (in **mme_main.sql**) must be modified to contain the filesystem mountpoint (by default, **/fs/ipod0**). For example:

```
INSERT INTO slots(path,zoneid, name, concurrency, slottype)
   VALUES('/fs/ipod0', 1, 'iPod', 1, 4);
```

Authentication chip interface

The **acp** option loads **iofs-***interface***-ipod.so** and checks if it supports an authentication chip interface. This option accepts one of the following values:

- **i2c**:[*options*] — authenticate in the iPod driver; see **iofs-i2c-ipod.so** for **i2c** options
- **cta** — authenticate over the serial pins and tell the iPod to grant authenticated privileges to the USB transport

See "Authenticating iPods" in the *MME Developer's Guide* chapter Working with iPods.

About filenames

With **long** filenames, **io-fs-media** requires a large cache and operations can be slow. In order to access a filename, **io-fs-media** may need to load into the cache the filenames for all files in a directory on the iPod (which can be in the tens of thousands), and do a string comparison on every filename, starting with the first filename, until it reaches the file it has been requested to find.

With **short** filenames, **io-fs-media** can go directly to the requested entry, and **devctl**s can be used to obtain the full filename and other information from the iPod. The operation is quick and does not require a large cache.

Restoring settings

The **restore** option is used to instruct the iPod to restore or keep the specified settings when it is removed from the system (unplugged). Valid settings are:

- **all** — restore all settings
- **eq** — change the equalizer setting

- **none** — restore no settings
- **repeat** — change the repeat setting
- **shuffle** — change the shuffle setting
- **sndchk** — change the sound check setting

Separate the settings you want to specify by colons ":". Precede settings you do *not* want to restore by an exclamation mark "!". For example:

- **io-fs-media -dipod,restore=all** — restore all settings
- **io-fs-media -dipod,restore=none** — restore *no* settings
- **io-fs-media -dipod,restore=!shuffle:repeat** — do not restore the shuffle setting, and restore the repeat setting
- **io-fs-media -dipod,restore=eq:sndchk** — restore the equalizer and sound check settings
- **io-fs-media -dipod,restore=eq:!sndchk** — restore the equalizer setting, and do not restore the sound check setting
- **io-fs-media -dipod,restore=all:!repeat:!random** — restore all settings *except* the repeat and random settings

Adding digital sampling rates

The **darates** option is used to add digital sampling rates the iPod driver requires to support digital audio. The "**+**" attribute adds the required 32000, 44100 and 48000 sampling rates. Optional sampling rates, such as 8000, 11025, 16000, 22500 and 24000, can be added by using the *rate* attribute, listing them with a colon ":" separating each item in the list. For example, to add the minimum three sampling rates required for digital audio, plus the optional 22500 and 24000 rates, start the driver as follows:

```
# io-fs-media -dipod,transport=usb,acp=i2c,darates=+22500:24000
```

Preferences

The **pref** option sets preferences on iPods and iPhones at startup. It requires:

- the preference class followed by a forward slash ("/")
- the setting, followed by a forward slash
- the restore setting: either, *k*, or *r*. See "Setting persistence" below.

Multiple preferences are separated by colons (":"). For example:

```
# io-fs-media -dipod,pref=video/on/k:ratio/wide/r
```

Supported classes and preference settings are listed in the table below:

Class	Settings	Behavior
video	off	Disable video output.
video	on	Enabled video output.
video	ask	Ask user if not in EI mode.
screen	fill	Fill entire screen.
screen	fit	Best fit while maintaining ratio.
format	ntsc	NTSC video format and timing.
format	pal	PAL video format and timing.
lineout	off	Line-out disabled (iPhones).
lineout	on	Line-out enabled (iPhones).
connection	none	No connection.
connection	composite	Composite video (interlaced).
connection	svideo	S-video (interlaced).
connection	component	Component Y/Pr/Pb (interlaced or progressive, based on model).
caption	off	Closed captioning disabled.
caption	on	Overlays closed captioning on video content *before* outputting.
ratio	full	Used when destination monitor has 4:3 ratio.
ratio	wide	Used when destination monitor has 16:9 ratio.
subtitle	off	Subtitles overlays are disabled.
subtitle	on	Overlays subtitle text on video *before* outputting.
audioalt	off	Alternate audio channel is disabled.
audioalt	on	Alternate audio channel is enabled.

Due to a current Apple firmware limitation, closed captions are not supported on iPod Classic devices.

Setting persistence

The **`pref`** option *k* and *r* values determine what the iPod does with the setting when it is disconnected:

- *k* — keep this setting as the default
- *r* — restore previous setting

Splash screens

You can use the **splash** option to specify one or more of the following for the iPod splash screen, using colons to separate the different image paths, in this order:

- a greyscale image (for older greyscale iPod devices)
- a color image
- a big color image

The order of the image paths is fixed; that is, the first path is always the path to the greyscale image file, the second path is always the path to the small color image file, and the third path is always the path to the big color image file. Thus, for example, to load all three image files, you would start the iPod driver with the **splash** option like this:

```
# io-fs-media -dipod,transport=usb,acp=i2c, \
    splash=path/greysplash.qnxlogo: \
    path/colorsplash.qnxlogo: \
    path/bigcolorsplash.qnxlogo
```

However, to to load only a large image file, you would start the iPod driver with the **splash** option like this, leaving the unused image paths empty:

```
# io-fs-media -dipod,transport=usb,acp=i2c, \
    splash=::path/bigcolorsplash.qnxlogo
```

Finally, if you do not want a big color image, simply omit the path; you do not need to add a colon to the end of your list. For example, to load only a small color image:

```
# io-fs-media -dipod,transport=usb,acp=i2c, \
    splash=:path/colorsplash.qnxlogo
```

Uploading a splash screen to an iPod

To upload a splash screen to an iPod:

1. Save the image you will use for the splash screen in the required format:
   - color — RGB-565 LE
   - greyscale — monochrome 2 bits per pixel
2. Add to the beginning of the image file, an 8-byte header with the information described in “Splash image file header” below.
3. Start the iPod driver with the **splash** specifying the path(s) to the splash screen image file(s).

Splash image file header

iPods expect splash image files to have an 8-byte header, as described in table below.

Bytes	Format	Description
0-1	16 bits, little endian	Width of the image, in *bits*.
2-3	16 bits, little endian	Height of the image, in *bits*.
4-7	32 bits, little endian	Stride of the image, in *bytes*

The *stride* of an image is the number of bytes used for each row in the image's file. For example, if an image is 15 bits wide, then its stride will probably (but not necessarily) be two bytes; that is 15 bits of image information, plus 1 bit of padding to byte-align the information.

For more information about iPod splash screen image formats, please refer to the Apple specifications.

iPod configuration selection at startup

iPods have a tendency to reset if you do not use the **-c** option when you start **io-usb**.

When you start **io-usb**, use the **-c** option so that the launcher application selects the iPod configuration to use, instead of just defaulting to the device's first configuration.

Examples:

Below are examples of how to start **io-fs-media** using the iPod driver for a serial connection and for a USB connection.

Serial connection

Start the iPod filesystem for an iPod with a serial cable connection:

```
# io-fs-media -dipod,transport=ser,acp=i2c
```

This command uses the default:

- serial port **/dev/ser1**

and is equivalent to:

```
# io-fs-media -dipod,transport=ser:dev=/dev/ser1, \
    acp=i2c:addr=0x10:path=/dev/i2c0
```

USB connection

Start the iPod filesystem for an iPod with a USB connection:

```
# io-fs-media -dipod,transport=usb,acp=i2c
```

This command is equivalent to:

```
# io-fs-media -dipod,transport=usb: \
    acp=i2c:addr=0x10:path=/dev/i2c0
```

See also:

io-fs-media, **iofs-i2c-ipod.so**, **iofs-pfs.so**, **iofs-ser-ipod.so**, **iofs-usb-ipod.so**, **mme**

Syntax:

io-fs-media -dpfs,*options*

Runs on:

ARM, PowerPC, SH, x86

Options:

The *options*, in addition to the standard **io-fs** options, include:

Standard options

debug=*n* Set the debug level.

mount=*path* Specify the root of the filesystem for PFS devices. If *path* ends in **'$#'**, then **'$#'** is replaced with a unit number. Example: **-dpfs,mount=/fs/pfs'$#'**

norefs Don't have objects with references appear as directories. This disables playlists (for example, **.alb** and **.pla** files on some devices), but can significantly reduce synchronization time.

device=*bus_no*:*device_no*:*interface_no*

Start the **iofs-pfs** module with one program instance per PFS device. See "Directed PFS device startup" below.

DRM options

The DRM options are optional. The **iofs-pfs** module supports DRM by default, and will function for non-DRM-protected playback even if the DRM support files aren't found on the target. The **cfile**, **kfile**, and **sfile** options are required only if you've changed the default location of these support files from **/etc/pfs/**.

drm Don't allow PFS operation without DRM support (initialization fails if no DRM files are found).

cfile=*filename* Specify the file containing a certificate to be used with DRM. The certificate file is obtained from Microsoft.

kfile=*filename* Specify the file containing an RSA private key to be used with DRM. The key definition is 580 bytes. This file is obtained from Microsoft.

sfile=*filename* Specify the file containing a serial number to be used with DRM. This file is expected to be 16 bytes in length. This file is obtained from Microsoft.

Directed PFS device startup

The **device** option allows the PFS driver to be started with one program instance per PFS device, rather than with a single program instance servicing multiple PFS devices.

To start the **iofs-pfs** module to support one PFS device per instance of **iofs-pfs**, use the **device** option and specify the paths for the bus, device and interface for each. For example, to handle two PFS devices (**device=***bus_no*:*device_no*:*interface_no*):

```
# io-fs-media -dpfs,device=1:3:3
# io-fs-media -dpfs,device=2:4:6
```

Bus, device and interface numbers are hexadecimal values.

Non-compliant device support options

These options allow you to customize how **iofs-pfs** interacts with legacy devices that don't fully support the PFS 2.01 specification.

For some of these "fix" options, you can apply the fix to all devices, specific devices (by specifying the vendor ID and device ID), all devices from a specific vendor ID (by specifying only the vendor ID), or a known list of devices that require the fix (by specifying **known**). In the options below, *VVVV* (vendor ID) and *PPPP* (device ID) are four-digit hexadecimal values. These options are additive, so you can use them more than once to add devices to the "fix" list. You can, for example, specify **getfix=known,getfix=0e791129** to apply the fix to the known set of devices plus device 0x0e791129.

force[=*VVVVPPPP* **|** **known**]

Force USB devices to be considered as MTP devices.

getfix[=*VVVVPPPP* **|** known]

Apply a fix to MTP-extended get-object processing, so that devices have the entire file read in at once (since they do not handle extended duration get object transactions very well). By default, such devices are not supported. The amount of the file that gets read in is determined by the **getsize** option.

getsize[=*size*]

Allocate a fixed size buffer to read in the entire contents of objects when the attached PFS device does not support the **GetPartialObject** MTP command. By default, such devices are not supported. The size of the buffer may be between 1MB to 256MB, with a default of 10MB if no size is specified. Files that exceed the size of the buffer may still be read, but the content of the file will be truncated to fit into the buffer.

getx [=*VVVVPPPP* **|** **known**]

Use the extended-duration get-object mechanism to read files on the specified devices that do not support the GetPartialObject

MTP command. If *VVVVPPPP* is not specified, then the extended duration processing will be used for all devices that do not support the GetPartialObject MTP command.

The **getfix** option overrides this option, since some devices cannot handle the extended-duration get-object operation.

noplist[=*VVVVPPPP* **| known**]

Do not use property list MTP commands since they do not work properly.

Description:

This **io-fs** module provides a media filesystem interface for Microsoft PlaysForSure (PFS) / Media Transport Protocol (MTP) devices.

Enabling Digital Rights Management (DRM)

The PFS module for the **io-fs** filesystem supports Microsoft's Windows Media DRM 10 for WMA content on Playsforsure devices.

When PFS starts, it looks for the files **cipher-aes.so** and **iofs-msdrm10.so** in the **lib/dll** directory that is referenced by **LD_LIBRARY_PATH**. In addition to the dynamic object library files, there are three configuration files required for DRM, which are provided separately by Microsoft. If these files cannot be found, then DRM will not work. By default, **iofs-pfs** looks for these files in **/etc/pfs**. You may specify an alternate location for these files with the **-kfile**, **-sfile**, and **-cfile** options for **io-fs**.

To force PFS to use DRM, specify the **drm** command-line option when you start **io-fs**. All required files for using DRM must be present on the target system (see the "Required Files" section below).

Required Files

These files are required in the **PATH** on the target system if you start **iofs-pfs** with DRM enabled:

lib/dll/cipher-aes.so

A library that supports AES decryption.

lib/dll/iofs-msdrm10.so

A library that is used by the PFS module to provide DRM functionality.

/etc/pfs/drm_certificate.bin

The binary image of a certificate. It is typically a text file that you must get from Microsoft. The name and location of this file can be changed by using the **-cfile** option to **io-fs**. For example:
-dpfs,cfile=/my_certificate.crt.

/etc/pfs/drm_serial_number.bin

This is a binary image of a 16-byte serial number. The serial number is embedded in a DRM message sent to PFS devices. You can change the name and location of this file by using the **-sfile** option to **io-fs**. For example: **-dpfs,sfile=/my_sn**.

/etc/pfs/drm_private_key.bin

A binary file that contains a definition of an RSA private key. You should specify as many fields as possible. You can change the name and location of this file by using the **-kfile** option to **io-fs**. For example: **-dpfs,kfile=/my_key.key**.

The format of the file is given by the following structure definition:

```
// this structure defines the contents of the PFS drm_private_key.bin file
// - contains information needed to define private key for RSA
// - all fields are big endian numbers (MSB first, LSB last)
// - all fields should contain appropriate values
struct RSA_binary_private_key_ {   // 580 bytes in total
    unsigned char bExponent[4];    // required: exponent for public key
    unsigned char bModulus[128];   // required: modulus for public key
    unsigned char bP[64];          // optional: prime number P
    unsigned char bQ[64];          // optional: prime number Q
    unsigned char bDP[64];         // optional: private exponent modulo P - 1
    unsigned char bDQ[64];         // optional: private exponent modulo Q - 1
    unsigned char bInverseQ[64];   // optional: inverse of prime number Q
    unsigned char bD[128];         // required: this is the private key exponent
};    // 580 bytes total
```

Files:

See Required Files for a list of files required when you enable DRM.

See also:

iofs-ipod.so, **io-fs-media**, **mme**

iofs-ser-ipod.so

The serial transport mechanism for connecting to iPods

Syntax:

io-fs-media -dipod,transport=ser:*options*

Runs on:

ARM, PowerPC, SH, x86

Options:

Options for **iofs-ser-ipod.so** are separated by colons (:). These options are:

audio=*devicepath* Set the path to the location from where **io-media** should read in the audio. See "Setting the audio path" below.

baud=*number*[-*number2*]

Set the baud rate. If a second number is specified, the highest baud rate is tried first, and the other number is a fallback value. The default value is **19200-57600**.

iPod devices officially support only two baud rates, 19200 (third-generation iPods) and 57600 (all other devices).

dev=*devicename* The serial device used to communicate with the iPod device. The default value is **/dev/ser1** .

nopoll Don't poll. By default, polling is on.

noreadloop Don't use a read loop. By default, this option is set so the driver does *not* use a read loop. This option disables some workarounds in the iPod serial transport, and is required for QNX Momentics release 6.3.2 and older. It is not required for release 6.4 and more recent.

use Print out a usage message to **stdout** and exit.

Description:

The utility **iofs-ser-ipod.so** provides the transport mechanism for connection to iPod devices through a serial connection. It is loaded by **iofs-ipod** when it is started with the transport mechanism set to **ser** (for serial devices).

Setting the audio path

The **audio** option sets the path to the location from where **io-media** should read in the audio. This option can be used for when iPod accessed through serial transport (**iofs-ser-ipod.so**) and for iPods accessed through USB transport (**iofs-usb-ipod.so**).

If you specify a path for the **audio** option, **io-media** reads the audio from the location specified by this path. For example, with serial audio, you can specify the audio path to the location of the serial iPod audio driver's capture device as follows:

```
# io-fs-ipod -dipod,transport=ser:dev=/dev/ser1:audio=/dev/snd/pcmC0D0c
```

When **io-media** needs to read in audio data (to push it out to the sound card output), it will read from this audio string.

Examples:

Start the iPod filesystem for an iPod with a serial cable connection:

```
# io-fs-media -dipod,transport=ser,acp=i2c
```

This command uses the default:

- serial port **/dev/ser1**

and is equivalent to:

```
# io-fs-media -dipod,transport=ser:dev=/dev/ser1, \
    acp=i2c:addr=0x10:path=/dev/i2c0
```

See also:

io-fs-media, **iofs-i2c-ipod.so**, **iofs-ipod.so**, **iofs-pfs.so**, **iofs-usb-ipod.so**, **mme**

iofs-usb-ipod.so

The USB transport mechanism for connecting to iPods

Syntax:

```
io-fs-media -dipod,transport=usb
```

Runs on:

ARM, PowerPC, SH, x86

Options:

Options for **iofs-usb-ipod.so** are separated by colons (:). These options are:

audio=*devicepath*
: Set the path to the location from where **io-media** should read in the audio. See "Setting the audio path" below.

busno
: The USB bus or chip to which the iPod device is connected.

config
: Configuration value of the iPod HID interface (iAP).

devno
: The USB device address, as assigned by **io-usb**.

iface
: Interface value of the iPod HID interface (iAP).

sconfig
: Select the iPod HID configuration. This option is only required if no other entity selects the configuration. See "Selecting the iPod configuration" below.

use
: Print out a usage message to **stdout** and exit.

Description:

The utility **iofs-usb-ipod.so** provides the transport mechanism for connection to iPod devices on a USB bus. It is loaded by **iofs-ipod** when it is started with the transport mechanism set to **usb** (for USB devices).

Authentication is required to use USB transport to control an iPod.

Selecting the iPod configuration

You can select the iPod HID configuration through the **enum-usb** configuration file, or use the **sconfig** option to tell **iofs-usb-ipod.so** to select the iPod HID interface. This option is applied only if no other entity selects the iPod device configuration to use.

iPods, such as Shuffles, that present themselves as only USB mass storage devices have only one configuration.

Directed iPod device startup

The iPod driver using the USB transport mechanism can be instructed to manage a specific device. This feature is designed to be used in an environment that has a driver launcher application that starts and stops services for removable devices: when running for a specified device, the iPod driver ignores device insertion and removal notifications from **io-usb**.

To start the **iofs-ipod** module to support a specific iPod device, specify the paths for the bus and the device, and the iPod HID configuration and interface. For example:

```
# io-fs-media -dipod,transport=usb:busno=0:devno=1:config=2:iface=2
```

Bus, device, configuration and interface numbers are hexadecimal values.

Setting the audio path

The **audio** option sets the path to the location from where **io-media** should read in the audio. This option can be used for when iPod accessed through serial transport (**iofs-ser-ipod.so**) and for iPods accessed through USB transport (**iofs-usb-ipod.so**).

If you specify a path for the **audio** option, **io-media** reads the audio from the location specified by this path. For example, with USB audio, you can specify the audio path to the location of the USB iPod audio driver's capture device as follows:

```
# io-fs-media -dipod,transport=usb,acp=i2c,audio=/dev/snd/ipod-0-0
```

When **io-media** needs to read in audio data (to push it out to the sound card output), it will read from this audio string.

Examples:

Start the iPod filesystem for an iPod with a USB connection:

```
# io-fs-media -dipod,transport=usb,acp=i2c
```

This command is equivalent to:

```
# io-fs-media -dipod,transport=usb: \
    acp=i2c:addr=0x10:path=/dev/i2c0
```

See also:

io-fs-media, **iofs-i2c-ipod.so**, **iofs-ipod.so**, **iofs-pfs.so**, **iofs-usb-ipod.so**, **mme**

io-media-generic

Neutrino multimedia playback controller

Syntax:

io-media-generic [*options***]**

Runs on:

ARM, PowerPC, SH, x86

Options:

-B	Don't detach to the background.
-c *file*	Use the options specified in the configuration file *file* rather than the built-in defaults.
-C	Print the built-in configuration to the standard output, and then exit.
-D	Log more debugging information.
-G *graph,options*	Pass *options* to *graph* (none are currently defined). This overrides the corresponding built-in default, as well as the corresponding setting in the configuration file, if specified with the **-c** option.
-M *module,options*	Pass *options* to *module*. This overrides the built-in default, as well as the corresponding setting in the configuration file, if specified with the **-c** option.
-o *options*	Apply the *options* to the global options. This overrides the built-in default, as well as the corresponding setting in the configuration file, if specified with the **-c** option.
-q	Log less debugging information.
-S	Log to *stderr* instead of **slogger**.

Description:

The **io-media-generic** controller is required by the MME to manage media playback. HMI applications that use the MME don't access **io-media-generic** directly. All playback and ripping requests are passed via the MME's API.

How to configure **io-media** and its modules, and the implications of different configurations is described under the following headings:

- **io-media** modules
- **io-media** global options
- **mmf_graphbuilder** options

- **mmf_trackplayer** options
- **cdda_trackplayer** options
- **trackcopier** options
- mmf options
- aoi options
- **damping_audio_writer** filter
- **io-media** configuration file
- Configuring **io-media** to use DLLs
- Configuring **io-media** optimizations

io-media modules

The **io-media-generic** controller presents a generic API to clients (such as the MME), but the implementation of its functionality is contained in various *modules*. At compile time, **io-media** can be configured to use different modules, depending on target hardware and the media types to be played. The default **io-media-generic** includes modules that use the Multimedia Framework 2 (MMF2, called simply MMF) for decoding and encoding media files. This decoding and encoding can be performed entirely by the software, by a combination of software and hardware.

At system startup, You can alter the behavior of the modules by changing default settings, either with the **-M** command-line option at system startup, or by changing the **io-media** configuration file.

For more information about specific modules and their options, see the relevant sections below.

If you require an alternative software framework for media playback, or have a hardware DSP solution on your target, you can use a customized variant of **io-media**. Contact your QNX sales representative for more information.

io-media global options

When a module creates a graph to play a media file, the graph spawns various threads to read, process, and play the media data. Global options are applied to all modules.

Global options that set priorities apply to all threads spawned by modules:

audio-hi-prio=*priority*

Set the highest priority level for audio graph threads; the default is 22. These threads are typically the writer threads, which pass PCM data to audio hardware, or encoded data to a DSP. To ensure smooth audio playback, other processes that may use the CPU for more than a few milliseconds at a time should not have their priorities set higher than the **audio-hi-prio** *priority*.

audio-mid-prio = *priority*

Set the mid priority level for audio threads; the default is 15. These are typically CPU-intensive threads, such as software decode.

audio-lo-prio = *priority*

Set the lowest priority level for audio threads; the default is 12. These are typically reader threads, where data is being buffered.

play-status-prio = *priority*

Set the priority level for time update events; the default is 14.

rip-prio = *priority*

Set the ripping process priority level; the default is 9.

Applying a resource to a filter

The ability to apply MMF resources to a specified filter rather than to the entire graph is implemented in the following **io-media** modules:

- **cdda_trackplayer**
- **trackcopier**
- **mmf_graphbuilder**

Depending on the graph, the **filter=** attribute can be set to one of **reader**, **parser**, **decoder**, **audio_writer**, **rawfile_writer**, or **wavfile_writer**.

The resource is applied to the filter after its input is attached, but before its output is examined, except in the case of writer filters, for which the resource is applied *before* the input is attached.

The **filter=** attribute is optional. If it is not specified, the resource is applied to the graph after it has been built, but before it is finalized.

mmf_graphbuilder options

The **mmf_graphbuilder** module builds graphs used by the **mmf_trackplayer** module.

The queue options are for two types of buffer queues:

- queue1 — memory used to buffer raw (compressed) data, for anti-skip or network latency protection.
- queue2 — memory used to buffer decoded data to the audio driver.

The **mmf_graphbuilder** module has these options:

queue1-size = *size*
queue2-size = *size*

The number of media buffers to queue. The default for queue1 is **49**, and for queue2 the default is **8**. You can reduce this number on a memory-constrained device, or increase it to increase the amount of data buffered to counteract network latency for streamed data. The size of each buffer allocated depends on the media type.

queue1-pre-hw = *percent*
queue1-pre-lw = *percent*
queue2-pre-hw = *percent*
queue2-pre-lw = *percent*

The queue "high water" and "low water" marks, expressed as a percentage between 0 and 100. The queue1 default is 80 for the high water, and 20 for the low water. The queue2 default is 90 for the high water, and 10 for the low water. When the percentage of buffers filled exceeds the high water mark, **io-media-generic** delivers an event to the MME, and continues to send this event until the percentage of buffers filled falls below the low water mark.

When the MME receives the high water event, it delivers the user event MME_PLAY_ERROR_INPUTUNDERRUN to the client application.

queue1-time = *time*
queue2-time = *time*

The maximum time to queue, in milliseconds. The default for queue1 is **12000**, and for queue2 it's 1000. If you have more buffers than required by this time, the remaining buffers are allocated but not used. If this time exceeds the amount of buffer space allocated, only the allocated buffers are used, and this value is ignored.

pre-buffer = *setting*

Determines how "next" tracks are buffered to reduce the time between tracks (that is, to implement "gapless playback"). For any setting other than **none**, **io-media-generic** notifies the client (**mme**) when it's finished reading a track into a buffer (but it's still playing), so that the client can request a new track to be played.

CAUTION: You should alter this setting only if you understand how your writer filter is implemented to build graphs, and how your audio driver/hardware handles multiple connections. Attempting to establish two writer filter connections to an audio driver that supports only one connection can produce unexpected results.

Track buffering is managed by a *media graph*. The **mmf_graphbuilder** module creates a new graph for every track it's requested to play. This option determines how the module builds the graph for the next track when the first graph has finished reading track

data, but hasn't yet finished writing the buffered track data to the audio output.

- **none** — The graph for a finished track is destroyed before the graph for the next track is built. With this setting, **io-media-generic** doesn't notify the **mme** when a track has finished reading.
- **stream** — Only the stream reader portion of a graph is built; this effectively confirms that the requested track exists.
- **parser** — The stream reader and parser portion of the graph is built. This setting is safe for audio drivers or hardware that accept only a single writer connection. Building this portion of the graph confirms that the format of the requested track is recognized by the filter and should be playable.
- **build** — The entire graph is built, but not finalized or started: its buffers have not been allocated, and no data is read.
- **start** — The entire graph is built, and started in a paused state. The track is read into a buffer. This achieves the shortest possible gap between tracks, but requires that the audio driver and/or hardware supports multiple writer connections.

The default is **none**.

transition = *trans*

This option determines what the module does with the graph for a previous track before starting the graph for the next track.

CAUTION: See the caution for the **pre-buffer** option above before altering this setting.

The *trans* setting can be one of:

- **none** — do nothing right away.
- **stop** — stop the previous graph before starting the next graph. This might be required to free the connection to the audio output on some devices.
- **destroy** — completely destroy the previous graph before starting the next one. This might be required to free the connection to the audio output on some devices, depending on the writer filter implementation.

The default is **none**.

If the *trans* setting is **none** or **stop**, the old graph is destroyed after the new graph has begun playing. These options provide a more rapid transition, but require sufficient memory to build a new graph before the memory allocated for the old graph's buffers has been freed.

resource This option sets a specific resource for a graph or, if specified, a filter. The resource settings can be:

- **filter** = **reader | parser | decoder | audio_writer | rawfile_writer | wavfile_writer**; see "Applying a resource to a filter" above.
- **name** = **"***resource_name***"** — the resource name
- **type** = *resource_type* — the resource type
- **value** = *value* — the value to set the resource to
- **optional** = **yes | no** — what to do if the resource named isn't available in the graph; if **no**, the graph isn't build, and **io-media** returns an error. If **yes**, the graph is built.

By default, one resource is set: MM_CDDA_READ_MAX_RETRY is set to 0 (zero), which disables automatic retries at the streamer level.

mmf_trackplayer options

The **mmf_trackplayer** module monitors and controls a graph after it has been built by **mmf_graphbuilder**. It interprets client requests and sends events to the client. It has these options:

skip-on-error = **no |** *milliseconds*[,*percent*[,*max_skips*]]

This option sets how the module handles read errors, as follows:

- If the option is set to **no**, the trackplayer reports an error to the client without attempting to skip forward.
- If all three arguments are set, the trackplayer skips forward *s* milliseconds on the first error, increases the skip forward time by *p* percent on each subsequent error, for *a* number of attempts. The default values are 200 milliseconds, 100 percent, and 10 maximum skips.
- If only the *milliseconds* is used, the trackplayer behavior is equivalent to following settings: *milliseconds*, *percent*,1. The trackplayer skips forward *s* milliseconds, once; then either reads successfully or fails. Since there is only one skip forward, the *percent* is irrelevant.
- If only the *milliseconds* and the *percent* are used, the trackplayer behavior is equivalent to the following settings: *milliseconds*, *percent*, *infinitely large number*. The trackplayer skips forward *s* milliseconds, increasing the skip time by *p* percent until either it is able to successfully read the track, or it skips beyond the end of the track.

cdda_trackplayer options

The **cdda_trackplayer** module contains functionality for playing audio CDs. It has these options:

skip-on-error = **no |** *milliseconds*[,*percent*[,*max_skips*]]

This option sets how the module handles read errors. For more information, see the **skip-on-error** option for **mmf_trackplayer** above.

queue-time The maximum time to queue, in milliseconds. The default for **queue-time** is **15000** milliseconds. If you have more buffers than required by this time, the remaining buffers are allocated but not used. If this time exceeds the amount of buffer space allocated, only the allocated buffers are used, and this value is ignored.

prefetch Prefetch pauses playback to allow media buffers to fill with sufficient data so that there are no interruptions to playback once it has started. This option specifies the prefetch behavior for the filter. If this option is set to 1 or 2, the behavior is applied only when an explicit "play", "change speed" or "seek-to-time" command is received from a client application, or when a track is started "gapless" when playback of another track is finished. Values can be:

- 0 — disable pre-fetch.
- 1 — delay playback until the buffer reaches the high-water mark (80 percent of buffer).
- 2 — delay playback until the buffer reaches the low-water mark (20 percent of buffer).
- 3 — delay playback if the buffer falls below the low-water mark *even if no explict command is received from the client application*, and resume playback when buffer reaches high-water mark.

The default value is 2: delay playback until the buffer reaches the high-water mark.

dts_writer = **"" |** [**"***module name***"**]

This option specifies the writer filter to use when playing a DTS CD. When this option is empty (**""**), the trackplayer will use the default writer filter, as specified by the **mmf audio_writer** option.

trackcopier options

The **trackcopier** module contains functionality for media copying and ripping. Its option is a list of filter resources to apply to the **mmf** graph after the graph is built:

resource This option sets a specific filter resource. The filter resource properties are:

- **name** = **"***resource_name***"** — the resource name.

- **type** = *resource_type* — the resource type; it must be one of **long**, **longlong**, **float** or **string**, and must match how the resource is defined by the filter.
- **value** = *value* — the value to set the resource to.
- **optional** = **yes | no** — what to do if the resource named isn't available in the graph; if **no**, the graph isn't build and **io-media** returns an error. If **yes**, the graph is built.

You can repeat the **resource** option as many times as necessary to list all the required filter resources. By default, one resource is set: MM_RAWFILE_WRITER_BUFFER_SIZE is set to **16384**, which specifies the buffer size the graph uses when writing the copied or ripped file.

mmf options

The **mmf** module contains MMF-specific functionality, and has these options:

audio_writer = *audio_filter*

This option sets the filter/DLL used by MMF graphs, such as **mmf_trackplayer** and **cdda_trackplayer** to write out audio information. The default **audio_writer** corresponds to the MMF **audio_writer.so** filter for standard audio output to a PCM device. You can change this setting to a different audio writer filter to support a DSP output, for example.

If you change the **audio_writer**, make sure the supporting DLL is available to **io-media**.

dlldir = *directory*

Set the location of the MMF filters. If this option is not specified, **io-media** uses the directory specified by the **$MM_INIT** environment variable. If this option is not specified in the command-line startup, and *not* specified in the **io-media** configuration file, **io-media** reports an error and exits.

keepdlls = *setting*

This option determines how **io-media** handles DLLs after they've been loaded. The *setting* can be one of:

- **none** — unload any DLLs when they aren't in use. A DLL is unloaded if the graph that was using it is destroyed, and the DLL is not being used by another graph.
- **used** — after a DLL is first used, keep it loaded forever.
- **all** — permanently load all DLLs from the specified directory.

The default is **none**.

CAUTION: It may not be safe to use the **used** or **all** settings with some filter DLLs, depending on how they are designed. It is safer to use the **aoi** module options to keep specific DLLs loaded.

utf8hook=*conversion_dll.so*

This option specifies a DLL to perform custom character conversions.

aoi options

This module contains Addon Interface functionality, used by the **mmf** module. You can force **io-media** to load a filter DLL and keep it memory using this option:

```
load {
    dll = /dll/media/somedll.so
    keep = "yes"
}
```

There is no corresponding **-M** command-line setting for this option; you need to use a configuration file to set it. See "Configuration File" below.

damping_audio_writer filter

The **damping_audio_writer** filter can fade the PCM stream IN or OUT.

To have this filter drop the volume level during trickplay mode by ramping down the PCM stream use, modify the **io-media** configuration file so that **io-media** MMF-based graphs use the **damping_audio_writer** filter as the audio output filter. For more information, see "Configuring output writer filters" below.

The following resources are supported:

TrickPlayVolumeFade

Trickplay volume percent to fade from total volume. Default is 0. Range is from 0 to 100 percent. Thus a value of 20 fades volume down to 80 percent.

TrickFadeTime — Fade IN/OUT time, in microseconds, applied during trickplay mode. Default is 20000 (20 milliseconds. Range is 2000-250000 microseconds (2 milliseconds to 250 milliseconds).

FadeTime — Fade IN/OUT time, in microseconds, applied during play mode. Default 2000 (20 milliseconds). Range is 2000-250000 microseconds (2 milliseconds to 250 milliseconds).

If you use the **TrickPlayVolumeFade** resource with the **mmf_graphbuilder** module, make sure that the *filter* attribute is set to "audio_writer". Below is a portion of an **io-media** configuration file showing how to set these resources:

```
module-options {
    module = "mmf_graphbuilder"
    ...
    resource {
        filter = "audio_writer"
        name = "TrickPlayVolumeFade"
        type = long
        value = 10
        optional = yes
    }
    resource {
        name = "TrickFadeTime"
        type = long
        value = 20000
        optional = yes
    }
    resource {
        name = "FadeTime"
        type = long
        value = 20000
        optional = yes
    }
}
```

- You only need the settings in the **io-media** configuration file: **io-media-*.cfg**, if you want to override the default settings for the resources.
- The **damping_audio_writer** filter has no effect on iPods because these devices control their own trick play behavior.

io-media configuration file

A copy of the relevant default configuration file is shipped with each **io-media** variant. You can copy and modify this file, then specify a configuration file to override **io-media-generic**'s built-in default settings using the **-c** command-line option.

The configuration file is a plain-text file with the following format:

```
io-media-config {
    # lines starting with a # character are comments
    # 1 global options section:
    global-options {
        # priority settings applied to all modules
    }

    # 1 or more module-options sections:
    module-options {
        module = module name
        # followed by module-specific settings
    }

}
```

This is an sample configuration file, showing the default settings:

```
io-media-config {
    global-options {
        audio-hi-prio=22
        audio-mid-prio=15
        audio-lo-prio=12
        play-status-prio=14
        rip-prio = 9
    }
    module-options {
        module = "cdda_trackplayer"
        skip-on-error = 200,100,10
        queue-time = 15000
        # Prefetch options.  0 = disable, 1 = resume on high water mark, 2 = resume on low
        # water mark, 3 = resume on high water mark, pause on low water mark.
        prefetch = 2
        # dts_writer, if not "", specifies a different writer filter for DTS CDs
        dts_writer = ""
    }
    module-options {
        module = "trackcopier"
        # MMF resources here
        resource {
            name = "MM_RAWFILE_WRITER_BUFFER_SIZE"
            type = long
            value = 16384
            optional = yes
        }
    }
        module-options {
        module = "mmf_graphbuilder"
        queue1-size = 49
        queue1-pre-hw = 80
        queue1-pre-lw = 20
        queue1-time = 12000
        queue2-size = 8
        queue2-pre-hw = 90
        queue2-pre-lw = 10
        queue2-time = 1000
        # pre-buffer can be "none", "stream", "parser", "build", or "start"
        pre-buffer = none
        # What to do with the old graph before starting the new one:
        # "none", "stop", or "destroy"
        transition = none
        resource {
            name = "MM_CDDA_READ_MAX_RETRY"
            type = long
            value = 0
            optional = yes
        }
        format {
            url = "cdda:*"
```

```
        parser = "cdda_parser"
    }
    format {
        url = "*.mp[a123]"
        parser = "mpega_parser"
        decoder = "xing_mpega_decoder"
    }
    format {
        url = "*.mp4"
        parser = "mp4_parser"
        decoder = "qnx_raac_decoder"
    }
    format {
        url = "*.m4a"
        parser = "mp4_parser"
        decoder = "qnx_raac_decoder"
    }
    format {
        url = "*.aac"
        parser = "aac_parser"
        decoder = "qnx_raac_decoder"
    }
    format {
        url = "*.wma"
        parser = "wma9_parser"
        decoder = "wma9_decoder"
    }
    format {
        url = "*.wav"
        parser = "wav_parser"
    }
}

module-options {
    module = "mmf_trackplayer"
    # If skip on error is not "no", it defines the number of milliseconds
    # to skip on a read error, the percentage to grow it by on each subsequent error,
    # and the maximum number of skips.
    skip-on-error = 200,100,10
    # Prefetch options.  0 = disable, 1 = resume on high water mark, 2 = resume on low
    # water mark, 3 = resume on high water mark, pause on low water mark.
    prefetch = 2
}
module-options {
    module = "mmf"
    audio_writer {
        url = "snd:*"
        filter = "audio_writer"
    }
    video_writer {
        url = "gf:*"
        filter = "gf_writer"
```

```
        }
        dlldir = "$MM_INIT"
        # keepdlls can be:
        #  "none" (unload any DLLs when they aren't in use)
        #  "used" (after a DLL is first used, keep it loaded forever)
        #  "all"  (permanently load all DLLs from the specified directory)
        keepdlls = "none"
    }
    module-options {
        module = "aoi"
        # For each DLL you want to load, specify a "load" element like this:
        # load {
        #     dll = "/dll/media/somedll.so"
        #     keep = "yes"
        # }
    }
}
```

Configuring output writer filters

The **mmf** module specifies the writer filters used to deliver audio and video streams to output devices. When an MMF-based **io-media** graph receives the URL of an audio or video output device, the graph selects a writer filter based on the output type (audio or video) and the URL of the device, using a list of *audio_writer* and *video_writer* elements in the **io-media** configuration file. Both types of element contain a pattern to match against the URL, and the name of the writer filter to use for devices that match the pattern.

If no matching element of the appropriate type is found for a device URL, the device is rejected, unless the (deprecated) *audio_writer* or *video_writer* attribute is also specified, in which case the value of this attribute names the filter to use.

For example, with the configuration shown below, assuming no other *video_writer* configuration elements or attributes, if the graph is asked to output to a video device whose URL matches the pattern specified by the *url* attribute ("gf:*"), it will use **gf_writer**; if no match is found the graph will reject the URL.

```
module-options {
    module = "mmf"
    ...
    video_writer {
        url = "gf:*"
        filter = "gf_writer"
    }
    ...
}
```

Updating the io-media configuration file

To update the **io-media** configuration file, after installing the latest release of **io-media**:

1. Save a copy of your **io-media** configuration file for later comparison.
2. Start **io-media** with the **-C** to print the default configuration file.
3. Save a copy of this file for future reference.
4. Refer to your old **io-media** configuration file, and edit the new configuration file to incorporate your project-specific settings.
5. After saving the new configuration file, restart **io-media** with the **-c** option and the path to the new configuration file.

Configuring io-media to use DLLs

To have **io-media-generic** use dynamic filters at least one of the following must be true:

- The **MM_INIT** environment variable is set.
- The DLL location has been specified by the **mmf** module's *dlldir* attribute in a configuration file or on the command line.

About permanently loaded filters and streamers

The default **io-media** configuration loads some filters and streamers permanently into memory; other filters are loaded and unloaded dynamically when necessary.

Some filters are loaded permanently because they are needed by the majority of graphs; if any of these required filters are missing, there is a high probability that **io-media** will not be able to play any media. These filters are, therefore, specified as required by the default configuration; if any of these filters are missing, **io-media** will exit at startup.

Some other filters are loaded permanently because they might be used for potentially high-bandwidth operations that could be significantly slowed down by having to load and unload filters. These filters typically support a particular operation or media type; if they are missing, **io-media** will not be able to play some media but will have no problems playing other media. Some of these filters are shipped in a separate, optional package, and may not be present on some targets. These filters are flagged as optional in the default configuration; if they are missing, **io-media** logs a warning to the system log, but does not treat the situation as a fatal error.

Configuring the set of required filters and streamers

You can change the minimum DLL set that **io-media** maintains in memory in order to make optimal use *for your environment* of the system resources you have available. Generally, the more DLLs in memory, the better the performance but the greater the memory required.

The **io-media** configuration file includes multiple *keepdll* elements under the **mmf** module's *module-options* element. Each *keepdll* element determines if the named filter or streamer is in the set or is optional.

The default setting is required: if it the DLL is missing, log an error and exit. Thus, in the example below:

- **tmpfile_streamer** is optional (because it is only used for playback during ripping) — it is not kept in memory, and **io-media** starts even if this streamer is missing
- **stream_reader** is required — it is kept in memory, and **io-media** will *not* start if this streamer is missing

```
module-options {
    ...
    keepdll {
        name = "tmpfile_streamer"
        optional = yes
    }
    keepdll {
        name = "stream_reader"
    }
    ...
}
```

Where to install io-media DLLs

To ensure system efficiency, all **io-media** DLLs should be installed in their own, exclusive directory, such as, for example, **/lib/dll/media**. No other libraries should be installed at this location, because **io-media** loads *all* DLLs from its DLL directory in order to check if it can recognize them as MMF filters or as other Addon Interface (aoi) DLLs. Placing **io-media** DLLs in a directory with other DLLs, such as for example, **/proc/boot**, may adversely affect system performance.

To set the directory **io-media** searches for its DLLs, use the *module-options* element's *dlldir* attribute in the **io-media** configuration file.

- If you set this element to **""**, **io-media** will *not* search any directory for DLLs.
- If you configure **io-media** to not search any directory for DLLs, you can still use the *load* attribute in the **io-media** configuration file to have **io-media** load specific DLLs.

Troubleshooting io-media DLL paths

If **io-media** does not find its DLLs it will fail to process the media files for which the missing DLLs are required. If this situation occurs in your environment, you may encounter symptoms such as the following, reported in the slog:

```
Jan 10 10:04:36    2    20     1
io-media-generic/trackplayer: Error 1
Jan 10 10:04:36    2    20     1
io-media-generic/trackplayer: Couldn't load the MediaReader filter
```

The information provided in the example above suggests the following problems:

- The **$MM_INIT** environment variable may be missing, or may not be set to the correct path; because **stream_reader** is the first entity the MMF tries to get when it builts a graph for **io-media**, the message "Couldn't load the MediaReader filter" suggests that **$MM_INIT** may not be set to the path for the directory with the MMF filters.
- **stream_reader** may be missing from the directory with the MMF filters.
- **libmmfilter.so.1** may not be in the library search path.

The MMF filters (files with **.so** extensions) used by **io-media** are typically installed at **lib/dll/mmedia**. When you start **io-media**, it needs to know the path to this directory. You can specify this path by one of the following methods:

- Set the value of the environment variable **$MM_INIT** to the full path to the directory with the filters.
- When you start **io-media**, use the **mmf** command-line option **dlldir** to set the path to the directory.

Other shared libraries must be in your library search path. This path is specified by the **$LD_LIBRARY_PATH** environment variable. Make sure that all shared libraries required by **io-media** are installed in a path specified.

- Depending on whether your **io-media** variant has streamers linked in statically or relies on DLLs, an incorrectly set **$MM_INIT** environment variable may produce a generic "Unable to open %s" error message instead of the MediaReader message in the example above.
- A filter (other than **stream_reader**) missing from the DLL directory may produce messages such as "Couldn't create a linking filter for *filter name* stream", or variations of the "Couldn't load the *filter name* stream filter" error message.
- The MMF, which builds graphs for **io-media**, doesn't tell **io-media** whether it failed to build a graph because a filter couldn't be loaded or because it didn't like the file it is attempting to process. If you suspect a configuration problem, you should test it by trying to process media files that you know are supported and not corrupt or damaged.

Configuring io-media optimizations

You can configure **io-media** for optimal performance when it selects filters for its graphs, when it encounters damaged (scratched) media, and when it attempts to retrieve metadata:

- Using the **io-media** *format* element and its *strict* attribute
- *StickyReadError* resource
- *QuickMetadataScan* resource
- Configuring the time **io-media** waits for metadata

Using the io-media *format* element and its *strict* attribute

io-media maintains a list of URL patterns that you can associate with filters, filter resources, and options, such as the *strict* attribute.

To associate filters, filter resources and options with a URL pattern, use the **io-media** configuration file's *format* element. The *format* element can have the following attributes:

url required
: This element defines a pattern to compare to the URL of the file being processed. If no *format* element's *url* attribute matches the given URL, none of the *parser*, *decoder*, or *strict* attributes are considered to be specified for the file.

parser optional
: This element specifies name of the parser filter to use to process the file.

decoder optional
: This element specifies the name of the decoder filter to use to process the file.

strict optional, but if it is specified this attribute must be set to **yes**
: This element instructs **io-media** to use *only* the specified parser or decoder, if a parser or decoder is specified.

resource optional: none, one or many may be specified
: This element specifies a resource to apply to the filter; see "Applying resources to afilter" above.

To build a graph for a media file, **io-media**:

1. Selects the first *format* element whose *url* attribute matches the URL for the file to be processed.
2. Attempts to open the file and build an MMF graph using the information provided in the format element.
3. If a parser is needed to build the graph (which is usually the case), and a parser is specified for the selected pattern, **io-media** tries that parser.

4. If the parser rejects the file, or if no parser was specified for the URL pattern:
   - If the strict option is not specified, **io-media** uses a rating-based query to find a parser: it attempts to find the best parser for the file by having all parsers examine the file, with each parser rating its ability to process the file's contents.
   - If the query fails, or if it is not performed because the *strict* attribute is set, then the graph-building fails.
5. If the parser accepts the file, processing continues.

If at a later point in the processing a decoder is required (for example, if a stream is encoded and the writer doesn't have a built-in decoder), a similar process is repeated to find a decoder.

If none of the patterns specified by *format* element *url* attributes matches the URL for the file to be processed, **io-media** uses its rating-based query to search for a parser or decoder, as required. That is, **io-media** proceeeds as though the last *format* element in its configuration file were configured thus:

```
format {
    url = "*"
    }
```

URL — pattern test order

When **io-media** tests for matches of URL patterns, it performs the test in the order of the *format* elements in its configuration file.

This order is relevant *only* in the case where more than one pattern could match a URL. In such cases, the most likely configuration would be to place the pattern least likely to match the URL first, since this pattern would typically describe a special case that should be overridden for URLs that match both patterns.

For example, you would list **/fs/cd0/*.mp3** before ***.mp3** in order to apply special settings for MP3 files on a slow CD-ROM, but apply other settings for all other MP3 files — files on USB sticks and hard drives. Placing the more general ***.mp3** pattern first would match *all* MP3 files, including those on CD-ROMs, and the special settings for these files would never be applied.

Using the *strict* attribute

The *strict* attribute causes **io-media** to fail immediately and return an error if:

- a specified parser or decoder fails to identify a file

  or

- a parser or decoder is required to process a file but is not specified for the selected URL pattern

To use the *strict* attribute, modify your **io-media** configuration file (**io-media-*.cfg**) to include "strict = yes" for each parser for which you want this attribute applied. For example:

```
format {

    url = "*.mp[a123]"
    parser = "mpega_parser"
    strict = yes
}
```

To speed up failure by rejecting files based on their extensions (i.e. *.pdf), for a specified format set the *strict* attribute and do not specify a parser. This configuration ensures failure except when the URL calls for a streamer that knows that the stream it must process is uncompressed audio.

A simple way to reject all files that don't have any of the recognized extensions is to add the following *format* element to your **io-media** configuration file:

```
format {
    url = "*"
    strict = yes
}
```

- The *strict* attribute is optional; it does not have to be included in your **io-media** configuration file. However, if you include it, it must be set to **yes**.
- To maintain consistent behavior across your system, use the same *strict* attribute for all your parsers.
- You may prefer inconsistent behaviour for your projects. For instance, you might prefer files on a CD-ROM to use a strict policy to speed up failures, but files on a hard drive or a USB stick to be probed more thoroughly because these devices are faster and don't get scratched as easily as CDs. To configure your system for this sort of behavior, you can split the format descriptions information in two, for example, by adding an entry for **/fs/cd?/*.mp[a123]** before the entry for ***.mp[a123]**.

Using the *StickyReadError* resource

The *StickyReadError* resource is implemented with the **fildes_streamer** AOI streamer. This resource reduces the time a client application has to wait for **io-media** to return with an error after it has encountered damaged or otherwise unreadable media. When this resource is set, a streamer error returned by any read, sniff or seek operation is sticky: after one of these operations fails, the streamer fails all subsequent attempts to perform these operations without actually attempting to read anything from the filesystem.

io-media should set the *StickyReadError* resource at graph creation time. If **io-media** is configured for skip-on-error behavior, it should reset the resource at

graph playtime, falling back to the skip-on-error configured behavior. If **io-media** is *not* configured for skip-on-error behavior, the *StickyReadError* resource can be maintained for the life for the graph.

The *StickyReadError* resource feature uses the **io-media** ability to apply MMF resources to a specified filter rather than to an entire graph. For more information, see "Applying a resource to a filter" above.

To configure **io-media** to use the *StickyReadError* resource, you must configure the **io-media** configuration file to tell **io-media**:

- how to handle read errors when parsing metadata (*value* is set to 1 so we don't get retries when parsing metadata):

```
module = "mmf_graphbuilder"
resource {
    filter = "reader"
    name = "StreamerStickyError"
    type = long
    value = 1
    optional = yes
}
```

- how to handle read errors during playback (*value* is set to 0, so that **io-media** skips over errors and retries, as specified by other **io-media** configurations):

```
resource {
    name = "StreamerStickyError"
    type = long
    value = 0
    optional = yes
}
```

Using the *QuickMetadataScan* resource

The *QuickMetadataScan* resource is supported by the **mpega_parser** filter. Setting this resource instructs **io-media** to skip time consuming tasks, such as seeking and scanning, when attempting to extract metadata from a file.

If *QuickMetadataScan* is set, when **io-media** is asked to play a file and extract its metadata, **io-media** does *not*:

- seek to the end of the file to look for a possible ID3V1 tag
- parse ID3V2.*x* tags larger than 63 kilobytes
- scan the file to get a better average bitrate; instead **io-media** uses one of the following methods:
  - get the bitrate from a **xing** header if this header is present, or
  - extrapolate the bitrate from the first MP3 frame

All **io-media** variants that load the **mpega_parser** filter for playback inherit the behavior described above.

To configure **io-media** to use the *QuickMetadataScan* resource, you must configure the **io-media** configuration under:

```
module-options {
    module = "mmf_graphbuilder"
    ...
}
```

For example, to set the *QuickMetadataScan* resource on any MP3 files coming from the device mounted at **/fs/cd0**:

```
format {
    url = "/fs/cd0/*.mp[a123]"
    parser = "mpega_parser"
    decoder = "xing_mpega_decoder"

    # You can set MMF graph-level parameters here:
    graphparam {
    name = "QuickMetadataScan"
    value = "1"
    }
}
```

Or, to set the *QuickMetadataScan* resource on all MP3 files, regardless of their location:

```
format {
    url = "*.mp[a123]"
    parser = "mpega_parser"
    decoder = "xing_mpega_decoder"

    # You can set MMF graph-level parameters here:
    graphparam {
    name = "QuickMetadataScan"
    value = "1"
    }
}
```

Configuring the time io-media waits for metadata

An **io-media** option allows you to set a maximum time **io-media** will wait for metadata from device, such as an iPod, that manages its own track sessions.

This option applies to the **mediafs_2wire** graph. The default waiting period is 200 milliseconds.

To configure this option, change the value of the **metadata-timeout** attribute in the **io-media** configuration file:

```
module-options {
    module =  "mediafs_2wire "
    metadata-timeout = 200
}
```

Configuring the reader timeout

io-media includes a configurable resource that sets the reader timeout interval, in microseconds.

To use this resource, update the **graphbuilder** configuration in the **io-media** configuration file, following the example below:

```
resource {
    filter = "reader"
    name = "MM_TMPFILE_STREAMER_READ_TIMEOUT"
    type = long
    #  This timeout is in microseconds:
    value = 10000000
    optional = yes
}
```

Configurable read timeouts for cdda_streamer

The **cdda_streamer** MM_CDDA_DMA_CAM_READCD_TIMEOUT resource allows you to configure the DMA read timeout period, in seconds. The timeout configured with this resource corresponds to the **devb/ATAPI** driver's *cdrom timeout* settings. MM_CDDA_DMA_CAM_READCD_TIMEOUT requires that DMA be enabled. Thus, it can be used if DMA is enabled on CDDA playback, or when performing a ripping operation (DMA is enabled with the MM_CDDA_DMA_CAM_READCD resource). If DMA is not enabled, **cdda_streamer** ignores this resource .

Configuring the resource

The MM_CDDA_DMA_CAM_READCD_TIMEOUT resource is set in the **io-media** configuration file.

If you do not specify the MM_CDDA_DMA_CAM_READCD_TIMEOUT timeout resource, or if you set the timeout to 0 seconds, DMA reads will time out according to the "group 1" timeout of the **devb** *cdrom timeout* settings. A non-zero timeout setting for the MM_CDDA_DMA_CAM_READCD_TIMEOUT resource sets the number of seconds before **cdda_streamer** times out on DMA reads.

- For the **cdda_trackcopier** graph, you *must* specify **optional = yes** in order to permit copying of files, such as MPEG3 files, from block devices such as USB sticks, data CDs, and HDDs.

Below is an example from a configuration file:

```
# Max retries when reading from CDDA:
# default value = 0 since devb driver
# typically does its own retries
resource {
    filter = "reader"
    name = "MM_CDDA_READ_MAX_RETRY"
    type = long
    value = 0
    optional = yes
}
# Enable DMA reads from CDDA:
# value = 0 DMA disabled, value = 1 DMA enabled
resource {
    filter = "reader"
    name = "MM_CDDA_DMA_CAM_READCD"
    type = long
    value = 0
    optional = yes
}
# Drive specific 'readcd.rsvd3'
# value to be used when doing DMA reads from CDDA
# Note: only applies if DMA is enabled
# (MM_CDDA_DMA_CAM_READCD), consult CD drive
# specs for value
resource {
    filter = "reader"
    name = "MM_CDDA_DMA_CAM_READCD_RSVD3"
    type = long
    value = 0
    optional = yes
}
# Timeout value in seconds when doing DMA reads from CDDA
# Value corresponds to devb driver
# 'cdrom timeout' values, value = 0 defaults
# to devb group 1 timeout. Note: only applies if DMA
# is enabled (MM_CDDA_DMA_CAM_READCD)
resource {
    filter = "reader"
    name = "MM_CDDA_DMA_CAM_READCD_TIMEOUT"
    type = long
    value = 7
    optional = yes
}
```

See also:

mme

The **io-media** section of the chapter Architecture, in *Introduction to the MME*

Syntax:

mcd *options** *config_file* **&**

Runs on:

Neutrino

Targets:

ARM, PowerPC, SH, x86

Options:

-D	Set the name of the mediastore list directory; the default is **.devices**.
-E	Set the name of the mediastore eject file; the default is **.eject**.
-I	Set the name of the mediastore insert file; the default is **.insert**.
-n	Set the subsystem mountpoint; the default is **/dev/mcd**.
-v	Increase the verbosity of messages written to **sloginfo**, from 0 to 7.
-V	Print output messages to console, as well as **sloginfo**.
config_file	The pathname to a required configuration file.

Description:

The **mcd** utility (media content detector, or MCD) monitors device and mediastore insertions and removals, and the presence of specified media content.

Overview

The MCD is a stand-alone utility for detecting devices, mediastores and specified media content. It is positioned between storage and USB device drivers, and any client application that needs to be informed of a device or mediastore activity, or of the presence of specified media types.

The MCD design separates the definition of actions conducted by the system from the implementation of these actions so that the actions can be easily edited or updated without changes to code.

MCD rules

The MCD provides a binary decision tree framework, applying rules and branching between rules according to match/fail results. These rules are used for detecting device and mediastore insertions and removals, and for classifying their content. They are specified in the MCD configuration file, and implemented with user callouts.

You can write rules for the MCD instructing it to monitor the presence or absence of any device, mediastore or file, as shown by the three examples below:

Monitor a mediastore

The rule below tells the MCD to monitor the physical insertion or removal of a CD-ROM mediastore on a device (the hardware) at the location **`/dev/cd*`**.

```
[/dev/cd*]
Callout=CD_MEDIA_IOBLK
```

Monitor namespace changes

The rule below monitors changes (mount or unmount) of a device or mediastore (such as a USB storage device) on the system. These sorts of changes usually indicate the physical insertion or removal of the device (the hardware) and its filesystem mounting or unmounting.

```
[/fs/usb*]
Callout=PATH_MEDIA_PROCMGR
```

Monitor the presence of files

The rule below polls the contents of **`/directory`** looking for files or directories that have been created in or removed from the directory being monitored. For example, typing **`touch /directory/file`** from a shell satisfies this rule, though no physical device has been inserted or removed.

```
[/directory/*]
Callout=PATH_MEDIA_SCAN
```

The MCD can be used as a framework from which to build an independent content detection system; that is, a content detection system that is independent of the MME.

MCD server

The MCD server:

- monitors a user-specified set of devices and their mediastores
- attempts to determine the category of media on mediastores (for example, an audio CD or a movie on a DVD)
- notifies clients of the presence of the media it has detected

In this documentation:

- *mediastore* is never a specific mediastore, but always *any mediastore of the specified type*: a CD and not, for example, the CD *I'm Your Man* by Leonard Cohen.
- an unformatted CD or USB stick is considered a device, because in its current state it cannot have any media content
- a USB stick with, for example, two partitions is one device with two mediastores

Operational flow

This section describes the MCD's operational flow at startup, and on detection of a new device, mediastore or file.

Startup

At startup, the MCD server proceeds as follows:

1 Read the configuration file.

2 Create a single, dedicated thread to provide the resource manager interface.

3 For each mediastore listed in the configuration file:

3a Create a dedicated, device detection thread.

3b Run the notification routine in its own thread.

Multiple detection threads, each for different mediastores, may be running concurrently.

Device or insertion

On detection of a device or mediastore insertion, or of the presence of a file of interest, the MCD proceeds as follows:

- Create a new thread to process the content detection rules for that device, mediastore or file.
- When the rule processing is complete, terminate the thread.

Configuring the MCD

The operation of the MCD is controlled by a configuration file. This file consists of named sections, each section defined by a name enclosed in square brackets: **`[ ]`**, followed by parameter lines with the form **`key = value`**. These parameters apply only to the section in which they appear.

The sample MCD configuration file **2phase.cfg** is delivered with the MCD; the sample configuration file **mcd.conf** provides examples for use with the Aviage Multimedia suite (MME).

The MCD ignores blank lines and any leading or trailing white spaces. It treats lines beginning with a "#" or a ";" character as comments and ignores them as well.

Configuration file sections

A section of the MCD configuration file can be one of:

- an entity (device, mediastore or file) description — the section name starts with a "/" character (i.e. **/dev/cd0**). See "Entity descriptions" below.
- a media content rule — any name without a leading "/". See "Media content rules" below.

The example below presents a description: **[/dev/cd0]** and a rule: **[DVD_AUDIO]**:

```
[/dev/cd0]
Callout     =   CD_MEDIA_IOBLK
Argument    =   1000,2000
Priority    =   11,9
Start Rule  =   DVD_OR_CD

[DVD_AUDIO]
Callout     =   FNAME_MATCH
Argument    =   /AUDIO_TS/AUDIO_TS.IFO
Match Rule  =   DVD_VIDEO
Fail Rule   =   DVD_VIDEO
```

Entity descriptions

For entity (device, mediastore or file) description sections, the section name is the entity the MCD monitors. This name can be a single name, such as **/media/drive**, or a wildcard pattern, such as **/dev/umass***. If the section name is a wildcard pattern, the event notification routine defined by the **Callout=** for the section must be capable of handling every entity that matches the wildcard pattern.

Parameters

Configuration parameters are used differently according to the type of section (mediastore description, or content rule) in which they are used.

Parameters in an entity section of the configuration file are used as follows:

Callout= The notification routine that the MCD runs when it detects the entity defined in the section name. Each notification routine is run in its own thread, allowing it to either block or poll as necessary.

If you provide no **Callout=** routine, you should handle device, file or mediastore transitions externally with the notification provided through the resource manager interface.

Argument= An optional argument to pass to the notification routine defined by the **Callout=** parameter.

Priority= The priorities at which to run, if applicable, for the entity defined in the current section:

- the content detection thread
- the notification thread

Start Rule= The root of the decision tree executed to determine media content type; that is, the first rule to apply to each entity following insertion notification.

Stop Rule= The root of the callouts to execute when the entity is removed.

Media content rules

For media content rule sections of the **mcd** configuration file, the section name is the name of the rule.

Parameters

Parameters in a media content rule section of the configuration file are used as follows:

Callout= The notification routine that the MCD runs when it detects an entity matching the content rule in the section name. See "Notification routine" below.

Argument= An optional argument to pass to the notification routine defined by the **Callout=** parameter.

Match Rule= The branch of the decision tree to execute when media content matches the content rule.

Fail Rule= The branch of the callouts to execute when no media content matches the content rule.

A rule runs at the priority given in the entity section that starts the rule chain.

Notification routine

The notification routine that runs when a mediastore matches a rule produces a match/fail result to indicate whether or not the media on the mediastore satisfies the routine's particular requirements. Based on the result produced by the rule, the MCD

takes a branch to another rule, as specified by the **Match Rule=** or **Fail Rule=** parameters in the current section.

If no associated branch rule is provided for a rule's result, the MCD considers the rule to be terminal and content detection complete.

If a rule contains no **Callout=** parameter, the MCD assumes that the rule either matches or fails, based on the presence of an associated branch rule. When debugging, you can use this characteristic to disable a test and always make a branch to the next rule.

Using the MCD as a filesystem automounter

The MCD can be used as a filesystem automounter by creating a set of two-phase rules in the MCD configuration file. Two-phase rules are implemented in the MCD as follows:

First-level entries

- The first level of entries in the MCD configuration file refers, *not* to mediastores, but to devices that can be monitored using the provided MCD callouts.
- The **Start Rule=** parameter in a first level entry points to the MOUNT_FSYS callout, a built-in routine that can mount filesystems based on simple mediastore criteria.
- Mounting a filesystem uses the triggers insertion notification of the second level of entries (for mediastores) via the PATH_MEDIA_PROCMGR built-in routine.

Second-level entries

- When they are triggered, the MCD configuration file's second-level entries perform the content detection algorithms on the mediastores at the filesystem mountpoint.

For an example of how to use the MCD as a filesystem automounter, see "Two-phase filesystem mount example" in the "Examples" section below.

The mcd resource manager interface

The MCD server presents a standard QNX resource manager (filesystem-like) interface. The default top-level directory is **/dev/mcd**; it includes:

- a set of **S_IFNAM/name-special**(**.insert** and **.eject**) files, which the MCD uses to provide a client API to the system
- a **.devices** directory with an entry for each entity known to the system

To change the top-level directory, use the **-n** command-line option.

`.insert` and `.eject` files

The **`.insert`** and **`.eject`** files are write-only files in **`/dev/mcd`**. External programs can trigger the MCD's content detection process on the insertion or ejection of an entity by writing to the appropriate file the pathname of the entity that has been inserted or ejected; for example **`/dev/cd1`** .

If the hardware for the device with an ejected mediastore doesn't support removal notification, then the MCD treats a subsequent insertion notification as an implicit ejection event.

`.devices` directory

The **`.devices`** directory in **`/dev/mcd`** contains an entry for each entity (device, mediastore and monitored file) known to the system. Each entity is represented by a **`S_IFCHR/char-special`** file in this directory. These files hold information about the entity in fields as follows:

- *st_mtime* — the time of the last state change for the entity.
- *st_ino* — the sequence number of the insertion. See "Sequence number" below.

An entry appears in **`.devices`** directory only for entities that have been inserted at least once. Since entity sections can be wildcards, a full list of potential entity matches can not be known in advance. The MCD can only know about an entity after it has been inserted. Thus, if a client application tries to *stat()* a particular device and fails with ENOENT, it should treat this failure as though the *st_ino* field is 0 (i.e., device ejected).

The "Client API" section below includes an example that illustrates how to use this interface.

Sequence number

The sequence number stored in *st_ino* for any entity (device, mediastore or file) can be either zero or non-zero. A value of zero means that the entity is not present in the system. A non-zero value means that the entity is present in the system.

At each re-insertion of the entity, the MCD increases the sequence number for that entity. Thus, for example, for a mediastore the values of *st_ino* might be in sequence: 1 (first insertion), 0 (removal), 3 (re-insertion), 0 (removal), 5 (re-insertion).

A client application can use the incrementing value of *st_ino* at each state change to check that it is up to date with, for example, a mediastore's activity after a series of rapid insertions and removals. For more information, see "Stale rules" below.

Example: Filesystem hierarchy

Below is an example filesystem hierarchy:

```
$ ls -al /dev/mcd
dr-xr-xr-x  1 root   root  11 Aug 02 19:46 .
n-w--w--w-  1 root   root   0 Aug 02 19:46 .eject
n-w--w--w-  1 root   root   0 Aug 02 19:46 .insert
nr--r--r--  1 root   root   0 Aug 02 19:46 CDDA_OR_DTS
nr--r--r--  1 root   root   0 Aug 02 19:46 CD_AUDIO
nr--r--r--  1 root   root   0 Aug 02 19:46 DVD_AUDIO
nr--r--r--  1 root   root   0 Aug 02 19:46 DVD_OR_CD
nr--r--r--  1 root   root   0 Aug 02 19:46 DVD_VIDEO
nr--r--r--  1 root   root   0 Aug 02 19:46 MIXED_AV
nr--r--r--  1 root   root   0 Aug 02 19:46 SVIDEO_CD
nr--r--r--  1 root   root   0 Aug 02 19:46 VIDEO_CD
```

Read-only entries for rules

The top-level **`/dev/mcd`** directory contains read-only entries for each rule defined in the configuration file.

Client applications can read from here the name of the device that satisfied a particular rule, with the read blocking until a device with content matching that rule is available. A non-blocking select and notify mechanism is also available to allow the client to wait on multiple rules, or to perform other work until a rule is triggered. Following the notification, the client application can read the rule to determine the device.

Callout templates

The MCD server provides a framework from which you can build a content detection system. Callout routines provide all the specific functionality in such a content detection system.

The MCD includes some common routines available for use where required in a static-linked library bound into the server. The MCD also supports extension routines provided by third-parties in DLLs dynamically linked at runtime. Thus, the system is extensible: if you require a new, unsupported detection test, you can implement it outside the server framework and ship it as a separate library.

In the content detection system configuration file, all **`Callout=`** items refer to a callout. These callouts are identified as internal or external by their names:

- If the function name contains an “at” (@) character (for example, **`myfunc@mylib.so`**), it refers to an external function in the named DLL, which is resolved at runtime.
- If the function name doesn’t contain an `@` character, the callout is an internal built-in routine.

Extension modules must include the MCD header file **<sys/mcd.h>** for appropriate manifests and type definitions.

Insertion and ejection notification

The prototype for media insertion notification callouts is:

void mcd_notify(char ***iomgr[2], char ***device**, void ***arg **);**

The MCD creates this routine in a dedicated thread, that should continually monitor the device. This thread should not return, *except* in the event of a serious error. If the thread encounters a serious error, it should set *errno* appropriately and return. On the return of an entity detection thread, the MCD will:

- log the error
- stop monitoring the entity whose monitoring thread encountered the error
- continue monitoring other entity detection threads

Arguments

iomgr An array containing the names of the interface entries where insertion and ejection events are reported. In a default system iomgr[0] = **"**/dev/mcd/.insert**"** and iomgr[1] = **"**/dev/mcd/.eject**"**, but the actual strings will reflect any command-line overrides.

device A pointer to the name of the device, mediastore or file to monitor.

device may be a wildcard, requiring the routine to monitor a group of devices, files or mediastores. When a device event occurs, the routine should write to the appropriate *iomgr[]* path the name of the specific device, mediastore or file that is affected by the event.

arg A pointer to a routine-specific argument. This routine-specific argument is provided as the **Argument=** parameter of the relevant device entry in the configuration file.

Built-in notification routines

The MCD's built-in media notification routines include:

CD_MEDIA_IOBLK

This routine interfaces the system to CD and DVD drives managed by the **devb/io-blk** filesystem. It exploits the filesystem feature of automatically invalidating open file descriptors upon media change: it opens the device block-special file and periodically polls it, treating any EBADF result as a transition indicator. It distinguishes between insertion and ejection by examining the advertised size of the device.

The **Argument=** option sets the polling periods (in milliseconds) at which to probe the file descriptor when the mediastore missing, and when it is present. The default for this argument is **"1000,2000"**, which corresponds to a one-second interval when the mediastore is missing, and a two-second interval when the mediastore is present.

USB_MEDIA_ENUM

Targets running QNX Neutrino 6.3.*n* releases only. This routine interfaces the system to the **umass-enum** utility (which is replaced by **enum-usb** in QNX Neutrino 6.4.0). It translates USB state change events from their native format into the insertion and ejection events expected by the USB enumeration server named by the **Argument=** option.

The **Argument=** option sets the name of the USB enumeration server to connect to (typically **"/dev/umass-enum"**).

PATH_MEDIA_PROCMGR

This routine connects the system to any device that dynamically attaches a resource manager pathname when the resource manager is present, and detaches the pathname when the resource manager is absent. It uses the *procmgr_event()* facility (available from QNX Neutrino 6.3.0 SP2 onwards) to receive notification of any changes to the global pathname space, then scans for the addition and/or removal of any device mountpoints matching the pattern defined by the callout name.

The **Argument=** option sets the name of the special directory where the OS **pathmgr** maintains mountpoints (typically **"/proc/mount"**).

PATH_MEDIA_SCAN

This routine scans for the presence of filenames that match the pattern defined by the callout name. It is similar to the PATH_MEDIA_PROCMGR routine, but since the creation and deletion of files doesn't trigger any filesystem events, this routine operates by scanning the specified directory at regular intervals.

The PATH_MEDIA_SCAN cause the MCD to behave differently, based on the presence or absence of a trailing "/" character at the end of the pathname, as follows:

- "/" present — send one notification when the pathname exists as a directory (including as the root directory of a mounted filesystem)
- "/" absent — send a notification for every matching filename pattern in the named directory

The **Argument=** option sets the poll period, in milliseconds, for scanning the directory.

Media content determination

The prototype for content detection rule callouts is:

```
int mcd_content( char *device, void *arg );
```

Arguments

device The name of the raw device or mediastore

arg A pointer to a routine-specific argument. This routine-specific argument is provided as the **Argument=** parameter of the relevant rule entry in the configuration file.

Returns

This routine returns values as follows:

- MCD_RULE_MATCHED — the rule is matched
- MCD_RULE_NO_MATCH — there is no match for the rule
- MCD_RULE_ABORT — there is a serious error; set *errno* and stop the remainder of the detection process for the device

If the routine invoked by the callout requires access to a filesystem on the device, it can use the DCMD_FSYS_MOUNTED_BY **devctl** to locate the appropriate mountpoint.

Built-in content detection rules

The MCD's built-in content detection routines include:

DVD_OR_CD This rule determines if a disk mediastore is a DVD rather than a CD (by issuing a **READ DVD STRUCT** command).

It ignores The **Argument=** option. The rule matches only if the media is a DVD.

CD_AUDIO This rule determines if the CD media has any audio content (by issuing a **READ TOC** command). It ignores the **Argument=** option. The rule matches only if the media contains audio tracks.

To facilitate the detection of mixed-mode and enhanced CDs that contain both audio and filesystem components, you can configure the CD_AUDIO rule as a non-terminal state; that is, with both the **`Match Rule=`** branch and the **`Fail Rule=`** branch provided. With the rule configured this way, after matching audio content, you can continue on with other rules to detect data content.

BLANK_CD — This rule determines if the CD media is blank or unrecorded (by issuing a READ DISK INFORMATION command). It ignores the **`Argument=`** option.

The READ DISK INFORMATION command and the physical detection of blank disks is only supported by newer CD-RW hardware, and will fail on older CD-ROM hardware. In fact, older hardware may not even detect the insertion of a blank or unrecorded disk.

FNAME_MATCH — This rule uses *fnmatch()* to match file pathnames on a device's filesystem. It requires that the inserted mediastore either be a filesystem, or have a filesystem mounted on it, because it is implemented with *access()* probing. The rule will automatically resolve to the filesystem level, if appropriate.

This behavior means that:

- If you are running this rule chain off a device that was a filesystem (i.e. **`[/fs/cd*]`**), this filesystem is the mountpoint at which the FNAME_MATCH rule will search for filenames.
- If you come to the FNAME_MATCH rule from a device (i.e. **`[/dev/cd0]`**), the rule will:
  1. Work out where **`/dev/cd0`** is mounted (probably **`/fs/cd0`**).
  2. Search that filesystem for the matching filenames.

The **`Argument=`** option is a comma-separated list of pathnames, based from the root of the filesystem. If the MCD finds any of the pathnames in the list on the mediastore, the rule is matched.

FNAME_PATTERN — This rule uses *fnmatch()* to match filename patterns on a mediastore's filesystem. It requires that the inserted mediastore either be a filesystem or have a filesystem mounted on it, because it is implemented with a *nftw()* traversal. The rule will automatically find any such filesystem.

The **`Argument=`** option sets a comma-separated list of patterns. If a filename matching any of the listed patterns exists in any directory on the filesystem, the rule is matched.

This option supports several other options that can be embedded in the listed patterns. For example: **`Argument = depth=2,*.c,*.h"`**. These "embedded" options are:

- **`basedir=`** — set the subdirectory at which to start the scan; the default is the root directory of the given entity
- **`depth=`** — set the maximum number of subdirectory levels to recurse into, (i.e. the maximum depth from the root); the default is 0, which means no depth limit

By default the scan for a pattern match is the entire target filesystem. You can use the **`basedir=`** and **`depth=`** options to direct and limit this scan.

MOUNT_FSYS

This rule is used to mount a filesystem onto a specified device, and is used to extend the MCD to operate as an auto-mounter.

The **`Argument=`** option sets the filename of a file of mount rules. Since this option is opened and parsed each time the rule is run, you should consider locating this filename on a ramdisk or in **`/dev/shmem`**.

This rule is typically used as the **`Start Rule=`** of a two-phase configuration, where the resulting mount operation triggers a PATH_MEDIA_PROCMGR action. The rules are processed from the file in order, stopping at the first (*fnmatch()*) match that either succeeds or specifies to skip the device (when the rule has only a pattern and no mount information). In order to select the appropriate filesystem, you can specify multiple rules for a removable device.

The file format is one rule per line, with each line containing fields separated by white spaces. For example:

```
#Device_pattern    Mount_point Fsys_type Mount_options
/dev/cd*           /fs/cd%#    udf       normv
/dev/cd*           /fs/cd%#    cd        normv,case=upper
/dev/umass*t1[124] /fs/usb%0   dos       fsi=use
/dev/umass*t[146]  /fs/usb%0   dos
/dev/hd*
```

The rules shown in the example above instruct the MCD to:

- Try to mount any CD first as a UDF filesystem.
- If mounting a CD as a UDF filesystem fails, try to mount it as ISO9660 filesystem.
- Ignore HD devices.
- Mount **`/dev/cd`***n* as **`/fs/cd0`***n*.
- Mount USB partitions as DOS to the first available filesystem mountpoint.

See "MOUNT_FSYS special sequences" below for more information about the mountpoint sequences.

The *normv* mount option works around OS namespace race conditions. It is only needed if the **mcd.mnt** is configured to try UDF then CD, and requires the "TC630SP2_1828_fsys-EIDE" PSP.

UNMOUNT_FSYS — This rule unmounts the filesystem from the mediastore specified in the rule name. It ignores the **Argument=** option.

This rule is typically used as the **Stop Rule=** of a CD_MEDIA_IOBLK mediastore that uses the MOUNT_FSYS action, when a mount would otherwise persist after the mediastore ejection. If the mediastore is presented by a **resmgr** that will exit or be terminated by an external manager (such as USB), then that presentation implicitly unmounts any relevant filesystem without the need for this rule. However, in most instances where a MOUNT_FSYS is used, you should also configure a matching UNMOUNT_FSYS in order to ensure that the filesystem for an ejected mediastore is duly unmounted.

MOUNT_FSYS special sequences

The MOUNT_FSYS rule uses special sequences, as follows:

- **%#** expands to the major device number of the mediastore. For example, with **/fs/cd%#**, **/dev/cd0** will be mounted at **/fs/cd0** and **/dev/cd1** will be mounted at **/fs/cd1**. **%#** does not permit multiple partitions; use **%0**.
- **%0** expands to a sequential, unique number. It is used principally to allow multiple USB partition rules in the **mcd.mnt**, all with the mountpoint names **/fs/usb%0**. The **%0** in the name causes the MCD to try allocate **/fs/usb0**, **/fs/usb1**, **/fs/usb2** and so on (starting form 0) until it finds a unique mountpoint name. For example, if there are filesystems already mounted at **/fs/usb0** and **/fs/usb1**, then the MCD expands **/fs/usb%0** to **/fs/usb2**.

Below is a sample **mcd.mnt** file that uses the **%#** and **%0** special sequences.

```
#-------------------------------------------------------
# Device                    Mountpt     Type  Options
#-------------------------------------------------------
/dev/cd*                    /fs/cd%#    cd    normv
/dev/umass[0-9]*            /           enum
/dev/umass[0-9]*t1[1234]    /fs/usb%0   dos
/dev/umass[0-9]*t1[1234].*  /fs/usb%0   dos
/dev/umass[0-9]*t[146]      /fs/usb%0   dos
/dev/umass[0-9]*t[146].*    /fs/usb%0   dos
/dev/umass*t7[789]          /fs/usb%0   qnx4
/dev/umass*t17[789]         /fs/usb%0   qnx6  sync=optional
/dev/umass[0-9]*            /fs/usb%0   dos
```

Client API

The MCD uses special rule entries created in the resource manager filesystem to notify client applications of media content matches.

A client application can call *open()* to access the rule entry in which it is interested, and when that rule is matched, it can then use *read()* to read from the entry the name of the relevant mediastore.

The *read()* function blocks until a match is made (unless *oflag* is set to O_NONBLOCK). For non-blocking notifications, use *ionotify()*. To wait on multiple rules, use *select()*.

Maintained information

In order to inform each client once and only once of each match, the MCD server maintains state information about each mediastore, matched rule, and client application.

When a new mediastore is inserted, any matched rule triggers notifications to the interested clients. If the mediastore was inserted before a client registered with the MCD, the first read the client makes is satisfied immediately. This behavior eliminates any start-up race conditions, such as, for example, there being media already in a drive at system startup, and the content detection process completing before the higher-level client applications are even started.

Example: Media player

A very simple media player might be designed as follows:

```
int     fd, cd;
char    device[_POSIX_PATH_MAX];

// Open the CD_AUDIO rule and wait for it to be matched.
fd = open("/dev/mcd/CD_AUDIO", O_RDONLY);
while (read(fd, device, sizeof(device)) != -1) {
    // At this point, device contains an audio CD ...
    cd = open(device, O_RDONLY);
    // ... read the toc, play it, etc.
    // Could monitor playback status with
    // DCMD_CAM_CDROMSUBCHNL.
    // If disk is ejected, this will fail.
    // Can loop back waiting for next insertion.
    // The rule will be re-armed for the
    // next match.
    close(cd);
}
close(fd);
```

Example: Polling

The **mcd** device **Start Rule=** and **Stop Rule=** rule chains are mutually exclusive: the ejection of a device cancels out inserted rules for that device (and vice-versa). Therefore, if you use *select()* or *ionotify()*, you should use them in conjunction with a non-blocking *read()*, as there is no guarantee that the notified state and/or rules of the trigger will remain valid (for example, if the media is ejected between the calls to *ionotify()* and *read()*).

The code snippet below illustrates one way to use *ionotify()* in conjunction with a non-blocking *read()*:

```
<PRE>
   fd = open(rulename, O_RDONLY | O_NONBLOCK);
   SIGEV_UNBLOCK_INIT(&evt);
   for (;;) {
       while (ionotify(fd, _NOTIFY_ACTION_POLLARM, _NOTIFY_COND_INPUT, &evt) != 0) {
           while (read(fd, device, sizeof(device)) > 0) {
               // 'device' matched on 'rulename'
           }
       }
       pause();
   }
</PRE>
```

In a real application, the event would likely be a pulse, and there would be an event-driven loop rather than a pause as in the code snippet above.

Stale Rules

Stale rules may occur if there is client decoupling, and/or a delay between the notification and the use of inserted mediastore; for example, due to the spawning of a separate media application.

To avoid stale rules, the MCD can include the mediastore's insertion sequence number with the rule notification, and applications can then match this number against the device entry in the **/dev/mcd/.devices** directory. If the device has been ejected since the rule was triggered, these values will no longer match, indicating that the rule no longer applies to the current device content, and that new rules may have been re-triggered.

If the client application requires an insertion sequence number, the MCD uses an XTYPE read to return an additional **uint32_t** of data with the mediastore name, and the _IO_XTYPE_MQUEUE message priority code, avoiding the need to make changes to the global **<sys/io_msg.h>** header file.

```
// Get rule notification using an XTYPE read
int        fd;
uint32_t   seq1;
char       device[_POSIX_PATH_MAX];

fd = open("/dev/mcd/CD_AUDIO", O_RDONLY);
_readx(fd, device, sizeof(device), _IO_XFLAG_BLOCK | _IO_XTYPE_MQUEUE,
```

```
    &seq1, sizeof(seq1));

// Open and check the current version of the inserted device
int         fd;
struct stat st;
uint32_t    seq2;
char        entry[_POSIX_PATH_MAX];

fd = open(device, O_RDONLY);

sprintf(device, "/dev/mcd/.devices/%s", device);
seq2 = (stat(device, &st) != -1) ? st.st_ino : 0;

// If these match, the CD_AUDIO rule is the same and still valid
// and 'fd' is open on that version of the media
if (seq1 == seq2) ...
```

Additional Information

This section describes how to use the MCD for specific operations:

- Detecting other kinds of system media
- Detecting USB and iPod devices
- Pattern matching and case-sensitivity
- Matching a single rule
- Detecting CD insertion with non-media content
- CD-Changer controlled by external firmware
- Using the MCD as a partition enumerator

Detecting other kinds of system media

To detect system media not handled by the routines included with the MCD:

1. Determine what distinguishes your new media type from all other media types. In many cases, the difference might simply be the presence of certain files or directories on the mediastore. For example, a navigation update disk might be identified by the presence of **`acios_db.ini`** or **`config.nfm`** files. In this case, a built-in routine such as FNAME_MATCH, could perform the detection; or you might have to write a custom routine and provide it to the MCD in an external DLL.
2. Determine the precedence to probe for this media amongst the existing media tests (first, last, after checking for audio content but before checking any other data rules, etc).

3. In the configuration file, make a new rule section for this test, with the appropriate **`Callout=`** rule, and splice it into the appropriate point of the decision tree by editing the **`Match Rule=`** or **`Fail Rule=`** parameters of both the preceding rule and the new rule. The name of the new rule can be used to trigger any client application when the new content is matched.

Detecting USB and iPod devices

The MCD can manage any kind of device, provided that a notification mechanism is available to report on insertions and start the detection process.

For USB devices, you can use the following entry in the MCD configuration file:

```
[/fs/usb*]
Callout     = PATH_MEDIA_PROCMGR
Argument    = /proc/mount
Priority    = 11,10
Start Rule  = ...
```

Targets running QNX Neutrino 6.3.*n* releases only. For USB devices, the **`umass-enum`** server in conjunction with the MCD's built-in USB_MEDIA_ENUM routine can provide the notification mechanism and start the detection process. Invoke **`umass-enum`** with the **`-r`** option to activate its resource manager interface, and use the following device entry in the MCD configuration file:

```
[/dev/umass/*]
Callout     =   USB_MEDIA_ENUM
Argument    =   /dev/umass-enum
Priority    =   11,10
Start Rule  =   ...
```

For iPod devices, the device entries dynamically attach pathnames (when these pathnames are present), and so can be handled with the MCD's built-in *PATH_MEDIA_PROCMGR()* routine. Use the following device entry in the MCD configuration file for iPods:

```
[/fs/ipod*]
Callout     =   PATH_MEDIA_PROCMGR
Argument    =   /proc/mount
Priority    =   11,10
Start Rule  =   ...
```

Pattern matching and case-sensitivity

The MCD's FNAME_MATCH routine attempts to access listed files by using the underlying filesystem, which applies any rules appropriate for that filesystem *in conjunction with* any specified mount options. Thus, case-sensitivity in pattern matching depends on:

- the built-in routine used
- the underlying filesystem

The table below lists the case-sensitivity and case-preserving characteristics of some common filesystems:

filesystem	case-sensitive	case-preserving
FAT	No	No
ISO-9660	No	No
Joliet	No	Yes
QNX4	Yes	Yes
RRIP	Yes	Yes
VFAT	No	Yes

Since the majority of mediastores used for multimedia storage are formatted for DOS/Windows use, it is likely that the filesystem will be case-insensitive. This case-insensitivity means that in any FNAME_MATCH rules the **`Argument=`** filename list can be given in either upper or lower case.

The FNAME_PATTERN routine processes directory entries from the filesystem through the **`libc`** *fnmatch()* function, which is case sensitive.

The directory output from each filesystem depends on whether it is case-preserving. If the filesystem is *not* case-preserving, then default rules are used to control the filename presentation. Refer to the **`cd case=lower|upper`** or the **`dos sfn=lower|upper|windows`** filesystem mount options.

Since the most common multimedia formats are case-preserving and will use the exact filename that a user or media application used to create their files, in any FNAME_PATTERN rules the **`Argument=`** pattern list should be given in both upper and lower case (as in the MIXED_AV rule in the Examples below.).

Matching a single rule

If you don't want multiple media types to match and only want to match the first rule, you can use the fact that if a matched rule has no **`Match Rule=`** branch the MCD stops its detection process.

In the configuration file, simply arrange the rules, from the **`Start Rule=`** through the **`Fail Rule=`** links, in priority order. Do not provide any **`Match Rule=`** branches. The MCD detection framework will test these rules in sequence until one is matched, then stop. For an example, see the VIDEO_CD and SVIDEO_CD entries in the Examples below.

Detecting CD insertion with non-media content

To be notified when a CD is inserted regardless of what content it contains, simply:

- Make a dummy rule with a no **`Callout=`** routine.

- Make a **Match Rule=** branch (so that it will always match when tested).
- Make this rule the **Start Rule=** of the device (with your dummy rule branching to the original start rule).

Your application can now block on that new rule, via the normal resource manager interface ("/dev/mcd/DISC_INSERTED"), waiting for device insertion. For example, one of the example configurations in this document could be modified as follows:

```
[/dev/cd0]
Callout      =    CD_MEDIA_IOBLK
Argument     =    1000,2000
Priority     =    11,9
Start Rule   =    DISC_INSERTED

[DISC_INSERTED]
Match Rule   =    DVD_OR_CD
```

CD-changer controlled by external firmware

To detect insertion events from a CD-changer that is controlled by external firmware (e.g. FJ-10), you should not use any of the buit-in MCD detection callouts, but trigger the insertion notification directly from the CD-changer controller stack.

Internally, all buit-in device detection callouts do their specific work, then write the name of the device to the special **/dev/mcd/.insert** entry. This behavior means that to detect insertions on changers controlled by external firmware, proceed as follows:

- In the device driver, wait for the CD-changer to be loaded and available for use by the system
- When the CD-changer is available, write the name of the device (as a string, i.e. **/dev/cd0**) to the MCD control point. Writing the name of the device to the control point triggers the rest of the content detection process.

In the configuration file, the relevant device entry would be like this (note that no **Callout=** is specified in this situation):

```
[/dev/cd0]
Priority     =    11
Start Rule   =    ...
```

The insertion notification code in the driver is basically this:

```
int    notify;

notify = open("/dev/mcd/.insert", O_WRONLY);
write(notify, "/dev/cd0", 8);
close(notify);
```

Similar code to handle ejections can be written to **/dev/mcd/.eject**.

Using the MCD as a partition enumerator

The MCD's MOUNT_FSYS rule can be used to determine if partition enumeration has occurred on a device (USB stick). This capability can be used to try to mount a filesystem on the raw device, if the device has not been partitioned.

In order to determine if partition has occurred, use the following configuration:

- Configure the USB enumerator to start **devb-umass** with the **blk auto=none** option; that is, to *not* automatically enumerate partitions.
- Configure the MCD device rule for the USB as follows:
  - it must be of the form **[/dev/umass*]**; that is, so that the pattern matches both the raw device and any partitions
  - its **Start Rule=** should be a rule that invokes MOUNT_FSYS
- Configure a second-phase set of mountpoint rules **[/fs/usb*]** to continue processing once a filesystem has been mounted from either a partition or a device.

  The **mcd.mnt** rule used by that MOUNT_FSYS should include the following:

```
/dev/umass[0-9]*         /           enum
/dev/umass[0-9]*         /fs/usb%#   dos
/dev/umass[0-9]*t1[124] /fs/usb%#   dos     fsi=use
/dev/umass[0-9]*t[146]  /fs/usb%#   dos
```

The control flow for this configuration is as follows:

1. USB stick inserted.
2. USB enumerator detects insertion and launches **devb-umass**.
3. **devb-umass** puts up **/dev/umass***X* pathname, triggering the MCD.
4. The MCD runs the MOUNT_FSYS rule.
5. If the media is non-partitioned:
   - **5a** The enum rule is executed and fails.
   - **5b** The code falls through and attempts **fs-dos** mount on raw device, which should succeed, resulting in the appearance of an **/fs/usb***.
6. If the media is partitioned:
   - **6a** The enum rule enumerates partitions and, thus, succeed, terminating the callout.
   - **6b** The enumeration makes **/dev/umass***X***t***N* names appear, re-entering the MCD device rule with a pattern that skips the enum rules, and instead tries **fs-dos** mounts on a partition, resulting in appearance of an **/fs/usb***.
7. Following either of the above two cases (5 or 6), MOUNT_FSYS is successful with a mount, and the MCD continues with **/fs/usb*** rules, typically some form of content detection or the triggering of a dummy INSERT rule.

X is the disk number (0, 1, 2, etc.). and *N* is the partition type (4, 11, 12, etc.); for example: **/dev/umass[0-9]*** or **/dev/umass[0-9]*t1[124]**. Thus, a path with **umass***X* refers to a device, while a path with **umass***X***t***N* refers to a partition.

Examples:

Consider the following sample configuration file for a CD-based system:

```
# Sample CD/DVD disk identification rules.

[/dev/cd0]
Callout     =   CD_MEDIA_IOBLK
Argument    =   1000,2000
Priority    =   11,9
Start Rule  =   DVD_OR_CD

[DVD_OR_CD]
Callout     =   DVD_OR_CD
Match Rule  =   DVD_AUDIO
Fail Rule   =   CD_AUDIO

[DVD_AUDIO]
Callout     =   FNAME_MATCH
Argument    =   /AUDIO_TS/AUDIO_TS.IFO
Match Rule  =   DVD_VIDEO
Fail Rule   =   DVD_VIDEO

[DVD_VIDEO]
Callout     =   FNAME_MATCH
Argument    =   /VIDEO_TS/VIDEO_TS.IFO
Fail Rule   =   VIDEO_CD

[CD_AUDIO]
Callout     =   CD_AUDIO
Match Rule  =   VIDEO_CD
Fail Rule   =   VIDEO_CD

[VIDEO_CD]
Callout     =   FNAME_MATCH
Argument    =   /VCD/INFO.VCD,/MPEGAV/AVSEQ01.DAT,/MPEGAV/MUSIC01.DAT
Fail Rule   =   SVIDEO_CD

[SVIDEO_CD]
Callout     =   FNAME_MATCH
Argument    =   /SVCD/INFO.SVD,/MPEGAV/AVSEQ01.MPG,/MPEG2/AVSEQ01.MPG
Fail Rule   =   MIXED_AV

[MIXED_AV]
Callout     =   FNAME_PATTERN
Argument    =   *.MP3,*.mp3,*.WMV,*.wmv,*.WMA,*.wma,*.AAC,*.aac,*.JPG,*.jpg,*.MPG,*.mpg
```

A single device, **/dev/cd0**, is monitored by the built-in *CD_MEDIA_IOBLK()* routine:

- When media is inserted, the content detection process begins with the DVD_OR_CD rule.
- If this rule matches (is a DVD) then the process branches to the DVD_AUDIO rule. If this rule fails, the process branches to the CD_AUDIO rule.

The DVD_AUDIO rule assumes the existence (using the built-in FNAME_MATCH test) of a **/AUDIO_TS/AUDIO_TS.IFO** file on the DVD indicating DVD audio content.

- Since both audio and video content may be present on a DVD, both Match and Fail branch from here to the same rule to try next: DVD_VIDEO.

  This behavior is similar to the behavior of the CD_AUDIO rule, as a CD can contain both audio and data content.

  Rules for determining CD data content, such VIDEO_CD or SVIDEO_CD, have only a Fail branch, since a match at these levels is mutually exclusive with any further content. If these rules match, then the content detection process stops.

- The final MIXED_AV rule has no branches. Regardless of the outcome, processing stops at this rule.

During the content detection processing phase, any clients that have registered against any matched rules will be notified. Multiple rules, or no rules at all, might be matched by an inserted CD: an enhanced audio CD with a bonus music video might match both CD_AUDIO and MIXED_AV rules, whereas a data CD backup of a development system would match none.

Two-phase filesystem mount example

Below is a configuration file involving USB devices; it requires that an external USB enumerator invoke **devb-umass blk auto=partition disk name=umass** in response to insertions.

```
[/dev/umass*t*]
Callout     =   PATH_MEDIA_PROCMGR
Argument    =   /proc/mount
Priority    =   11,10
Start Rule  =   MOUNT

[MOUNT]
Callout     =   MOUNT_FSYS
Argument    =   /dev/shmem/mcd.mnt

[/fs/usb*]
Callout     =   PATH_MEDIA_PROCMGR
Argument    =   /proc/mount
Priority    =   11,10
Start Rule  =   MIXED_AV

[MIXED_AV]
Callout     =   FNAME_PATTERN
Argument    =   *.MP3,*.mp3,*.WMV,*.wmv,*.WMA,*.wma,*.AAC,*.aac,*.JPG,*.jpg,*.MPG,*.mpg
```

Note that the device pattern specified in the example above avoids the raw device itself and only applies to partition entries. The mount configuration **/dev/shmem/mcd.mnt** referred to contains:

```
/dev/umass*t1[124]      /fs/usb%#   dos
/dev/umass*t[146]       /fs/usb%#   dos
```

See also:

mme, Configuring Device Support in the *MME Configuration Guide*.

mme-generic

MME resource manager

Syntax:

mme
[-c *config_file***] [-o** *logging***=***level***:***flags***] [-DFSv]**

Runs on:

ARM, PowerPC, SH, x86

Options:

-c *config_file*
: Load the specified configuration file; see below.

-D
: Print messages sent to the system log to the standard output. Using the **-v** option to set the verbosity level.

-F
: Stay in the foreground. By default, **mme-generic** is spawned into a different process. When it is in the foreground, you can use ctrl-c to kill it.

-o *logging*="*module@level*:*flags*"
: Set the logging verbosity levels for the specified modules. See "Setting log verbosity levels" below.

-S
: Log synchronization statistics to **dev/slog**.

-v
: Turn on verbose mode. In this mode, messages about MME activity are sent to the system log. Increase the number of **v**s to increase the verbosity. Use the **-D** option to also view the messages on the standard output.

-V
: Print to **stderr** the schema versions **mme** expects and the schema versions it finds. See "Checking schema versions" below.

Description:

The **mme-generic** is a resource manager that handles device discovery and synchronization. It also provides the high-level API for managing playback (play, stop, and seek commands), and manages the library database.

Before you start **mme-generic**, your target must also be running:

- the **tmp** module of **io-fs-media**
- **qdb**, with the MME configuration file
- **io-media-generic**
- **mcd**

See also the chapter MME Quickstart Guide in the *Introduction to the MME*.

Configuration File

You can start **mme-generic** with a configuration file to change its default behavior. To create a configuration file, you should edit the sample **mme.conf** file provided with the MME. This is a self-documented XML file, with all possible parameters commented out, so you can un-comment and modify the parameters you wish to change.

For more information about configuring **mme-generic**, see the *MME Configuration Guide*.

Checking schema versions

At startup, **mme** checks the schema versions installed and compares these to the versions it requires:

- If the expected and required schema versions match, **mme** runs and does not print any schema information.
- If the expected and required schema versions don't match, **mme** exits with exit code 2. Mismatched expected and found schema versions produce a response such:

```
# mme-generic 2>/tmp/ver.txt
# echo $?
2
# cat /tmp/ver.txt
Database Schema versions mismatched
File:mme.sql Database:main expected:16 got:19
File:mme_library.sql Database:mme_library expected:4 got:17
File:mme_temp.sql Database:mme_temp expected:4 got:4
```

- If schema version information is *not* found, then **mme** prints the message "version not found", and exits with the same error code as for mismatched expected and required schema versions.

Finding out the schema versions mme expects

You can use the **-V** option to find out the schema versions **mme** expects — afer an upgrade, for example. If you use the **-V** when starting it, **mme** just prints to **stderr** and exits. Correct schema versions produce a response like:

```
# mme-generic -V
Database Schema versions expected
File:mme.sql Database:main expected:16
File:mme_library.sql Database:mme_library expected:4
File:mme_temp.sql Database:mme_temp expected:4
```

Setting log verbosity levels

The **-o** option can be used to set logging verbosity levels for specified **mme** modules. It can be used to set, for example, a lower verbosity for synchronizations than for the rest of the MME so that, when detailed logging is required to debug or fine-tune some

other feature, the writing of synchronization logs for large mediastores does not increase synchronization times.
Specify the logging module, the logging verbosity level and the logging flags for each module separately, separating the options for the different modules with semi-colons. For example:

```
# mme-generic -c $QNX_TARGET/etc/mme.conf -o logging="sync@2:0;mdi@2:1;mdp@
```

Logging modules

The strings that identify **mme** logging modules are:

String	Module
imgprc	image processing module
mdi	metadata interface module
mdp	metdata plugin module
sync	synchronization module
mme	all other modules

The above list is not definitive. The logging modules may change. To find out what logging module strings are valid, call *mme_get_logging()* with the string referenced by the *name* argument set to NULL.

Logging flags

The logging flags are bit masks that configure logging behavior:

Value	Behavior
1	Also write anything logged to standard output.
2	Write timing logs.

Log verbosity levels can be retrieved and set with the *mme_get_logging.html()* and *mme_set_logging.html()* functions.

Files:

/db/*.sql	Supporting schema files for databases created by **qdb**
mme.conf	Sample configuration file.

See also:

mmecli

mmebrowse

MME browse client

Syntax:

mmebrowse [-m *device***] [-M** *msid***] [-q** *database***]**

Runs on:

ARM, PowerPC, SH, x86

Options:

-m *device*	The device to browse. If not specified, the default is **/dev/mme/default**.
-M *msid*	The MSID to start browsing. If this option isn't specified, you can set the MSID from the command-line once **mmebrowse** is running.
-q *database*	The MME database. If not specified, the default is **/dev/qdb/mme**.

Description:

The **mmebrowse** is a command-line utility that allows you to browse devices using *directed synchronization*. For more information about directed synchronization, see "Types of Synchronization" in the chapter Synchronizing Media of the *MME Developer's Guide*, and the *mme_sync_directed()* function in the *MME API Library Reference*.

When you start **mmebrowse**, it displays a help message listing all available commands. You can enter "h" at any time to view the help message again.

See also:

mme, **mmecli**

Syntax:

```
mmecli

mmecli [-c cpname] [-d dbname] [-est]
       [-i interval] [-r mask] command command_args
```

Runs on:

ARM, PowerPC, SH, x86

Options:

-c *cpname*	Set the MME control context to connect to. By default, **mmecli** uses **/dev/mme**.
-d *dbname*	Set the database device name. By default, **mmecli** connects to the database defined by the **QDBC_DBNAME** environment variable.
-e	Run in event mode (not interactive). In this mode, **mmecli** runs in the background and prints out received events to the standard output. This may be useful for monitoring or debugging other MME client applications.
-i *interval*	Set the time notification interval in milliseconds. By default this is 100 milliseconds.
-r *mask*	Register for events mask value. By default this is **0xFFFF**.
-s	Script mode (interactive). See below.
-t	Report MME_EVENT_TIME. If this option isn't set, this event is ignored.

Description:

The **mmecli** is a command-line client to the MME that allows you to issue commands corresponding to MME API function calls, and view events generated by the system. You must have the MME and required applications running to use **mmecli**. The source of **mmecli** is provided in the QNX Aviage multimedia suite as well, and contains examples of how to call most of the MME API functions.

Script mode

Script mode can be used to run **mmecli** in a shell script. All typical **mmecli** commands can be issued, as well as some additional script-specific commands. Before commands are evaluated, they are passed through a pre-processor that can expand placeholders with values (see the table below). Lines beginning with the # character are considered comments, and are ignored.

If any command fails, **mmecli** stops evaluating any more commands and exits with a failure return code.

The table below lists script mode placeholders:

Placeholder	Meaning
%t	The last created tracksession.
%f	The *fid* value returned by the last **getfid** command.
%c	The current control context ID.
%b	The last created bookmark ID.
%r0 - %r9	User registers.
%z	The last created zone ID, or the last zone ID returned by **play_get_zone**.
%V	The volume received from the last **getvolume** command.
%M	The mute value received from the last **getvolume** command.
%B	The balance value received from the last **getvolume** command.

If you want to use a **%** character in script mode, you need to escape it by using **%%**.

Script mode commands

These commands are available only in script mode:

.waitforevent *event* [*event2*...]

This command cause the **mmecli** to block waiting for any of the specified events to arrive. Events are categorized as "good" (prefixed with a **+**) or "bad" (prefixed with a **-**). When a good event arrives, **mmecli** unblocks and resumes executing the script. When a bad event arrives, **mmecli** unblocks and exits with a failure.

Because events are queued, its important that you flush the event queue for events you are waiting for that have been previously delivered. See the **.flushevents** command to do this.

You can wait for all events available in the MME's event header file (**<event.h>**). Use the full event name, minus the MME_EVENT_ prefix.

Example:

```
.waitforevent +TRKSESSION -PLAY_ERROR
```

.flushevents Flush all events from the client queue. This command returns after all events are flushed.

.echo [*message*] This command prints a message to **stdout**. This is useful for debugging.

.qdb *command* Execute a query against the database. This command succeeds as long as the underlying *qdb_*()* calls succeed. The resulting statement is printed.

.qdb_require_rows *command*

Execute a query against the database. If no rows are resulting from a request, then the command fails.

.setint 0-9 *value*

Sets the specified register (r0 to r9) to the specified integer value.

.delay *millisecond_value*

Delays execution for the specified number of millieseconds.

.expecterror [*errno*]

Expect an error on the next command. This will cause **mmecli** to treat the next command with inverted logic: if it returns -1, it passes, but if it returns ≥, it fails. Optionally, *errno* can be set also as a requirement when you are expecting a particular *errno* back. If any *errno* other than the one specified is returned, then the command fails.

Supported commands

The **mmecli** supports the following commands.

Note that where a command takes an SQL *statement*, or a parameter that contains more than one word, the string should be enclosed by either single (**'**) or double (**"**) quotation markses. Quotation marks within a string must be escaped. For example, if you want to create a new tracksession with the statement **select fid from library where filename = 'chord.wav'**, you would need to use this command:

```
newtrksession l "SELECT fid FROM library WHERE filename = \'chord.wav\'"
```

audio_get_status Get the audio status; see *mme_audio_get_status()*.

bookmark_create *name*

Create a *bookmark*; see *mme_bookmark_create()*.

bookmark_delete *fid bookmarkid*

Delete a bookmark based on fid or bookmarkid; see *mme_bookmark_delete()*.

button *button* — Respond to button events for navigable tracks; see *mme_button()*.

charconvert_setup — Indicate the default character encoding; see *mme_charconvert_setup()*.

copy add *statement* — Add a media copy operation; see *mme_mediacopier_add()*.

copy clear — Clear all files from the media copy queue; see *mme_mediacopier_remove()*.

copy cleanup — Clean up partially copied or ripped files; see *mme_mediacopier_cleanup()*.

copy disable — Disable the mediacopier; see *mme_mediacopier_disable()*.

copy enable — Enable the mediacopier; see *mme_mediacopier_enable()*

copy mode [background | priority]

Get the selected media copying or ripping mode. If you specify a mode, the mode is set. See *mme_mediacopier_get_mode()* and *mme_mediacopier_set_mode()*.

copy remove *statement*

Remove files from the media copy queue; see *mme_mediacopier_remove()*.

directed_sync_cancel *operation_id*

Cancels a specified directed synchronization; see *mme_directed_sync_cancel()*.

delete_mediastores *flags*

Prune unavailable mediastores; see *mme_dvd_get_disc_region()*.

dvd_get_disc_region *msid*

Get the region permissions for a DVD-video; see *mme_dvd_get_disc_region()*.

dvd_get_status — Get the status for a DVD; see *mme_dvd_get_status()*.

get_api_timeout_remaining

Get the number of milliseconds remaining before the API call times out; see *mme_get_api_timeout_remaining()*.

getautopause	Get the autopause mode; see *mme_getautopause()*.
getccid	Get the control context ID for the currently connected control context; see *mme_getccid()*.
getclientcount	Get the number of clients connected to a control context; see *mme_getclientcount()*.
getfid	Get the current playing *fid*; see *mme_play_get_info()*.
getlocale	Get the preferred language code; see *mme_getlocale()*.
get_logging	Get the log verbosity levels for specified logging modules; see *mme_get_logging()*.
getnumsyncing	Get the number of synchronizing devices; see *mme_sync_get_status()*.
getrandom	Get the random playback mode for a control context; see *mme_getrandom()*.
getrepeat	Get the repeat playback mode for a control context; see *mme_getrepeat()*.
getscanmode	Get the scan mode for a control context; see *mme_getscanmode()*.
getstatus	Get the status of the current track; see *mme_play_get_status()*.
get_title_chapter	Get DVD title and chapter information for the currently playing DVD track; see *mme_get_title_chapter()*.
getvolume [*outputdeviceid*]	Get the current volume setting; see *mme_play_get_output_attr()*.
isplayingnavigable	Indicates whether the playing media can be navigated (with the **`button`** command); see the MME_PLAYSUPPORT_NAVIGATION flag in **`mme_play_info_t`**.
isplayingvideo	Indicates whether the current track has video; see the MME_PLAYSUPPORT_VIDEO in **`mme_play_info_t`**.
lib_column_set *msid column value*	Set values in specified table column; see *mme_lib_column_set()*.
media_get_def_lang	Get the preferred media playback language; see *mme_media_get_def_lang()*.

media_set_def_lang *lang*

Set the preferred media playback language; see *mme_media_set_def_lang()*.

metadata_create_session

Create a new metatdata session; see *mme_metadata_create_session()*.

metadata_free_session *session_id*

End a metadata session; see *mme_metadata_free_session()*.

metadata_getinfo_current *session_id* [*groups*]

Get metadata for the currently playing track; see *mme_metadata_getinfo_current()*.

metadata_getinfo_file *session_id msid file* [*groups*]

Get metadata for a specified file, based on the filepath; see *mme_metadata_getinfo_file()*.

metadata_getinfo_library *session_id fid* [*groups*]

Get metadata for a specified file, based on the file ID; see *mme_metadata_getinfo_library()*.

metadata_image_cache_clear *msid*

Purge images from the image cache; see *mme_metadata_image_cache_clear()*.

metadata_image_load *sessionid md_rid image_index*

Load an image for a file; see *mme_metadata_image_load()*.

metadata_image_unload *sessionid img_rid*

Clear image from temporary storage; see *mme_metadata_image_unload()*.

metadata_set *fid* [*flags*] [*numstring*] [*year*] [*month*] [*mday*] [*track*] [*disc*] [*string*]
Set the metadata for a file; see *mme_metadata_set()*.

ms_clear_accurate *msid*.

Mark library entries for the mediastore *msid* as inaccurate; see *mme_ms_clear_accurate()*.

ms_metadata_get *msid filepath types*

Get metadata from a file; see *mme_ms_metadata_get()*.

mute [*outputdeviceid*] *mute*

Set the mute state (1=muted, 0=not muted); see **`mme_play_set_output_attr`**.

newtrksession *mode statement*

Create a new tracksession based on an SQL *statement*. The *mode* can be **l** for "library" or **f** for "file". See *mme_newtrksession()*.

next — Skip to the next track; see *mme_next()*.

output_set_permanent *outputdeviceid permanent*

Set the permanency status of an output device (where *permanent* can be 0=No, 1=Yes). See *mme_output_set_permanent()*.

pause — Pause the current playback; see *mme_play_set_speed()*.

play [*fid*] — Play the specified *fid*, or the current tracksession, if no fid is specified; see *mme_play()*.

play_at *offset* — Start playback at the specified offset; see *mme_play_offset()*.

play_attach_output *zoneid outputdeviceid*

Attach an output to a zone; see *mme_play_attach_output()*.

play_bookmark *bookmarkid*

Start playback from a bookmark; see *mme_play_bookmark()*.

play_detach_output *zoneid outputdeviceid*

Detach an output from a zone; see *mme_play_detach_output()*.

play_file *msid filename*

Play a track on an unsynchronized mediastore; see *mme_play_file()*. Deprecated: use file-based track sessions; see "Creating and modifying file-based track sessions" in the *MME Developer's Guide*.

play_get_speed — Get playback speed and direction (forward, reverse, pause) for tracks; see *mme_play_get_speed()*.

play_get_zone — Get the zone ID used by a control context; see *mme_play_get_zone()*.

playlist_create *msid name*

Create a new playlist; see *mme_playlist_create()*.

playlist_delete — Delete a specified playlist; see *mme_playlist_delete()*.

playlist_generate_similar

Generate a playlist like an existing playlist; see *mme_playlist_generate_similar()*.

playlist_set_statement

Set the SQL statement to create a playlist; see *mme_playlist_set_statement()*.

playlist_sync — Synchronize a specified playlist; see *mme_playlist_sync()*.

play_resume_msid *msid*

Set the track session ID to use when resuming playback of a mediastore; see *mme_play_resume_msid()*.

play_set_speed *speed*

Set the playback speed and direction, where *speed* is expressed in units of 1/1000 of normal speed. See *mme_play_set_speed()*.

play_set_zone *zoneid*

Set the zone ID used by a control context; see *mme_play_set_zone()*.

prev — Skip to the previous track; see *mme_prev()*.

resume — Resume paused playback; see *mme_play_set_speed()*.

resync_mediastore *msid* [*folderid*] *passmask*

Re-synchronize a mediastore; see *mme_resync_mediastore*.

rmtrksession *trksessionid*

Remove a track session from the database; see *mme_rmtrksession()*.

seek_title_chapter — Seek to a specified title and chapter on a track or mediastore; see *mme_seek_title_chapter()*.

seektotime *milliseconds*

Seek to a time in a playing track; see *mme_seektotime()*.

set_debug *verbose debug*

Set MME debug settings; see *mme_set_debug()*.

set_api_timeout *milliseconds*

Set the API timeout, in milliseconds; see *mme_set_api_timeout()*.

setautopause 0|1 — Set the autopause mode, where 0=off, 1=on; see *mme_setautopause()*.

setbalance [*outputdeviceid*] *balance*

Set the ouput balance; see *mme_play_set_output_attr()*.

set_files_permanent *0|1 fid_select*

Set files as permanent (1) or prunable (0); see *mme_set_files_permanent()*.

setlocale *locale* — Set the preferred language for media with unknown language; see *mme_setlocale()*.

set_logging [*name*] *verbose flags*

Set the verbosity levels for specified logging modules; see *mme_set_logging()*.

set_msid_resume_trksession *msid*

Set the track session ID to use when resuming playback of a mediastore; see *mme_set_msid_resume_trksession()*.

setpriorityfolder *folderid*

Set a priority folder for synchronization; see *mme_setpriorityfolder()*.

setrandom *mode* — Set the random playback mode, where *mode* can be 0=off, 1=all, 2=albums; see *mme_setrandom()*.

setrepeat *mode* — Set the repeat playback mode, where *mode* can be 0=off, 1=single, 2=all; see *mme_setrepeat()*.

setscanmode *milliseconds*

Set the scan mode and time; see *mme_setscanmode()*.

settrksession *trksessionid*

Set the current track session; see *mme_settrksession()*.

setvolume [*outputdeviceid*] *volume*

Set the output volume; see *mme_play_set_output_attr()*.

shutdown — Shut down the MME; see *mme_shutdown()*.

start_device_detection

Start device and mediastore detection; see *mme_start_device_detection()*.

stop — Stop playback; see *mme_stop()*.

sync_cancel *msid* — Cancel mediastore synchronization for the specified mediastore. If *msid*=0, synchronization is cancelled for all mediastores. See *mme_sync_cancel()*.

sync_directed *msid path passmask* [*recurse*]

Start a directed synchronization. If *recurse*=1, a recursive synchronization is performed. See *mme_sync_directed()*.

sync_file *old_fid new_msid* [*filename*]

Synchronize a specified file; see *mme_sync_file*.

sync_get_status *msid*

Get the status for the specified *msid*, or for all mediastores if *msid*=0; see *mme_sync_get_msid_status()* and *mme_sync_get_status()*.

sync_set_debug *verbosity*

Set the verbosity level for synchronization operations; see *mme_sync_set_debug()*.

trksession_append_files

Append tracks to a file-based track session; see *mme_trksession_append_files()*.

trksession_get_info Get information about the current track session; see *mme_trksession_get_info()*.

trksession_resume_state

Resume playing a track session at the last saved position; see *mme_trksession_resume_state()*.

trksession_save_state

Save the playing position of the current track session; see *mme_trksession_save_state()*.

trksessionview_get_current

Get information about the current track in the track session; see *mme_trksession_get_current()*.

trksessionview_get_info *track title*

Get information about the specified track in the track session; see *mme_trksession_get_info()*.

trksessionview_metadata_get *track title types*

Get metadata for a track in a track session; see *trksessionview_metadata_get()*.

trksessionview_read [*offset*] [*nfids*>]

Update the information in the **trksessionview** table; see *mme_trksessionview_readx()*.

trksessionview_update *track title*

Update the information in the **trksessionview** table; see *mme_trksessionview_update()*.

trksessionview_writedb

Write the MME track session view to the database; see *mme_trksessionview_writedb()*.

video_get_angle_info *title*

Get the video angle; see *mme_video_get_angle_info()*.

video_get_audio_info *title*

Get the audio information for video playback; see *mme_video_get_audio_info()*

video_get_info Get information about a video; see *mme_video_get_info()*.

video_get_status Get the video status; see *mme_video_get_status()*.

video_get_subtitle_info *title*

Get subtitle information for a video title; see *mme_video_get_subtitle_info()*.

video_set_angle_info *title index*

Set the video angle; see *mme_video_set_angle()*.

video_set_audio *title index*

Set the audio stream for video playback; see *mme_video_set_audio()*.

video_set_properties *display_mode zoom_mode width height*

Set the properties of a video; see *mme_video_set_properties()*.

video_set_subtitle *title index*

Set the subtitle for video playback; see *mme_video_set_subtitle()*.

zone_create *name* Create an output zone; see *mme_zone_create()*.

zone_delete *zoneid* Delete an output zone; see *mme_zone_delete()*.

Examples:

This is an example shell script that uses the **mmecli** script mode to play all the available media (you will need to change */path_to/mmecli* to the path to **mmecli** on your system):

```
#!/path_to/mmecli -s
.echo "START: play all script"

.echo "Creating new track session"
.flushevents
newtrksession l 'SELECT fid FROM library ORDER BY fid'
```

```
settrksession %t
.waitforevent +TRKSESSION

.echo "BEGIN: Testing .setint"
.setint 0 %t
.echo "END: Register 0 == %r0"

.echo "BEGIN: Playing at beginning"
play 0
# track change is ok, play error is not
.waitforevent +TRACKCHANGE -PLAY_ERROR
getfid
.echo "Playing fid %f"
.echo "Verifying that fid %f is in the nowplaying table"
.qdb_require_rows "SELECT fid FROM nowplaying WHERE fid=%f AND ccid=%c;"
.echo "Metadata for currently playing track"
.qdb "SELECT * FROM nowplaying WHERE ccid=%c;"

.echo "STOP:  done play all script"
```

See also:

mme

Syntax:

```
mmexplore [-d metadata] [-m device] [-f filter [regex] ]
          [-M msid] [-o display file] [-q database]
          [-r resolve] [-s sort] [-w window]
```

Runs on:

ARM, PowerPC, SH, x86

Options:

-d *metadata*
: The metadata to display. Valid values are **none**, **all** or **name**. If not specified, the default is **name**. If this option isn't specified, you can set the metadata to display from the command-line once **mmexplore** is running.

-f *filter* [**regex**]
: How to apply the regular expression to use when filtering explored files. Valid values are:

- **none** — don't filter
- **pass** — equivalent to MME_EXPLORE_FILTER_INCLUDE: include files that match the expression
- **block** — equivalent to MME_EXPLORE_FILTER_EXCLUDE: exclude files that match the expression

The expression must be enclosed in double quotation marks. For example: **# mmexplore -f "pass .mp3$"**. If this option is not specified, the default value is **none**.

-m *device*
: The device to browse. If not specified, the default is **/dev/mme/default**.

-M *msid*
: The mediastore ID (MSID) to explore. If this option isn't specified, you can set the MSID from the command-line once **mmexplore** is running.

-o *display file*
: Set the display file for output. Valid values are **stout**, **none** or a valid filepath and filename. If this option is not specified, the default value is **stout**.

-q *database*
: The MME database. If not specified, the default is **/dev/qdb/mme**.

-r *resolve*
: Flag instructing **mmexplore** to resolve playlists. Valid values are **all** and **none**. If not specified, the default is **none**.

-s *sort*
: Data or metadata to use to sort explored files. Valid values are:

- **none** — don't sort
- **filename** — sort by filename
- **name** — turn on metadata extraction to get the track (song) name, and sort by this field

If this option is not specified, the default value is **none**.

-w *window* — The size of the window (number of explored items) to display. If not specified, the default is 20.

Description:

The **mmexplore** is a command-line utility that allows you to explore mediastores using the MME's explore API. For more information about the MME's explorer API, see the chapter Unsynchronized Media in the *MME Developer's Guide*, and the relevant functions in the *MME API Library Reference*.

The QNX Aviage multimedia suite includes the source for this utility.

Supported commands

The **mmexplore** CLI supports the following commands.

b — Go back one explored level in the display window.

d *metadata* — Set or change the metadata to display; see *mme_explore_info_get()* in the *MME API Reference*. Valid values are **none**, **all** or **name**.

e *item* — Set the offset at which to start displaying items in the display widow. The **mmexplore** displays the path, offset and number of child items found for the path. For example:

```
Entering item "/xplore/gracetest" (offset 1).
Item '/xplore/gracetest' has 663 child items.
Display range (x,y) or all (a) or none (n)?
```

Enter the range of items to display (for example, **1,20** for the first 20 items), **a** for all items, or **n** for no items.

g *flag* — Instruct **mmexplore** to resolve the contents of playlists. Set to **1** to resolve playlists, or to **0** to not resolve playlists. The default is **0**.

You need to resolve playlists to see or play their contents.

m *msid* — Start exploring the mediastore specified by the path in *msid*, and create and set a tracksession for the items in the window; see *mme_explore_start()*, *mme_newtrksession()* and *mme_settrksession()*.

n *path* — Enter the path to an item on a mediastore.

p *offset*	Play the track specified by *offset* in the display window. At present, **mmexplore** plays only individual tracks, as specified by this command.
q	Quit **mmexplore**.
r	Refresh the contents of the explorer display window.

Display codes

The table below describes the display codes **mmexplore** uses to identify items it lists in its window. It places these codes in the second column of its item lists.

Code	Meaning
D	Folder (Directory)
DP	Directory Playlist (folder); the item is a folder that is actually a playlist. PFS devices provide these types of playlists.
F	File
P	Playlist (file)
PF	Playlist entry converted to a Filename
PI	Playlist Item (not converted to a filename)

Examples:

The sample below shows a short **mmexplore** session:

```
# mmexplore

MME Media Store eXplorer Example
h Show this information.
m <n> Select media store ID <n>.
e <n> Explore item indexed as <n>.
n <path> Explore item named <path>.
b Go up to parent item.
p [<n>] Play folder or file index <n>.
r Redisplay current view.
d <type> Set metaData type 'all', 'none' or 'name' (default).
g 0|1 Get flags; 1 means resolve playlists directly.
q Quit.

Media Stores:
Rows: 2  Cols: 3
Names: +msid+name+mountpath+
00000: |1|HardDrive|/|
00001: |2|RALLY2|/fs/hdumass10-dos-1|

# m 2

Exploring media store ID 2.
Item '2' has 3 child items.
1 F  <unknown>         /=RALLY2
```

```
2 D  <unknown>         /xplore
3 D  <unknown>         /mp3

# e 2

Entering item "/xplore" (offset 2).
Item '/xplore' has 2 child items.
1 D   <unknown>        /xplore/gracetest
2 F   <unknown>        /xplore/typescript

# e 1

Entering item "/xplore/gracetest" (offset 1).
Item '/xplore/gracetest' has 663 child items.
Display range (x,y) or all (a) or none (n)?
# 1,10

Displaying all items from start to item 10.

1 F  Bye Bye Bye            /xplore/gracetest/('N Sync) No Strings Attached [01] Bye Bye Bye.mp3
2 F  Voices Carry           /xplore/gracetest/('Til Tuesday) Voices Carry [05] Voices Carry.mp3
3 F  Back To Paradise       /xplore/gracetest/(.38 Special) Flashback The Best Of .38 Special [01] Back To ...
4 F  Hold On Loosely        /xplore/gracetest/(.38 Special) Flashback The Best Of .38 Special [02] Hold On ...
5 F  Back Where You Belong  /xplore/gracetest/(.38 Special) Tour De Force [02] Back Where You Belong.mp3
6 F  I'm Not In Love        /xplore/gracetest/(10cc) The Original Soundtrack [02] I'm Not In Love.mp3
7 F  Blackmail              /xplore/gracetest/(10cc) The Original Soundtrack [03] Blackmail.mp3
8 F  Brand New Day          /xplore/gracetest/(10cc) The Original Soundtrack [05] Brand New Day.mp3
9 F  Ambitionz Az A Ridah   /xplore/gracetest/(2Pac) All Eyez On Me (Book One) [01] Ambitionz Az A Ridah.mp3
10 F  Dancing Queen         /xplore/gracetest/(ABBA) Gold [01] Dancing Queen.mp3

# p 10

Playing folder "/xplore/gracetest" item 10.

# q
```

See also:

mme, **mmecli**

Index

!

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U