QNX® Aviage Multimedia Suite

MediaFS Developer's Guide

For QNX® Neutrino® 6.4.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 April 30, 2009.

Contents

About this Guide

The *MediaFS Developer's Guide* presents how the media filesystem (MediaFS) module expects device drivers to describe media devices and mediastores, and the *devctl()* messages that these drivers need to support.
This *Guide* is intended for:

- developers who design and write device drivers for use with MediaFS
- developers who integrate support for these devices into higher-level applications that use the MediaFS interface — applications such as the QNX® Aviage Multimedia Suite's Multimedia Engine (MME)

For more information about the MME, see *Introduction to the MME* and the other books in the MME documentation set.

The table below may help you find what you need in this book:

For information about:	See:
The MediaFS standardized interface	MediaFS Overview
The structure of MediaFS, and required POSIX function support	MediaFS Structure
MediaFS entities, including files, directories and symbolic links	MediaFS Entities
Media changer presentation to MediaFS	Media Changers
How to present playback states and controls to MediaFS	Managing Playback
Device management messages supported by MediaFS	Device Management Messages
Playback and status update supported by MediaFS	Playback Messages
Metadata retrieval messages supported by MediaFS	Metadata Messages
Playback structures and constants used by MediaFS	Playback Structures and Constants
How to retrieve album art	Getting Album Art
MediaFS events and their structures	MediaFS Events
Examples of code used to work with MediaFS	Appendix A: Examples

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 Technotes	MME technical notes.
MME Utilities	Utilities used by the MME.
MME Configuration Guide	How to configure the MME.
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

continued...

Reference	Example
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.

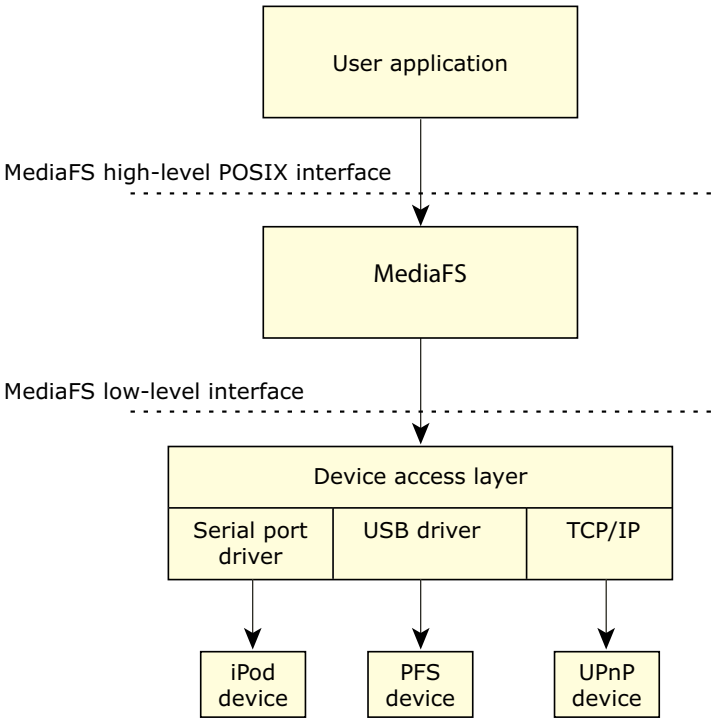
Chapter 1

MediaFS Overview

MediaFS presents a POSIX-compliant filesystem view of media devices. This filesystem view of media devices can be used by higher-level applications, such the MME, to browse and control media devices. These higher-level applications can use the MediaFS standarardized interface to query and control media playback on a wide range of media devices, including portable music devices such as iPods and PlaysForSure devices, and UPnP devices that attach to a network.

For more information about the MME, start with *Introduction to the MME*.

The following diagram shows the MediaFS module in relation to the user application and media devices.



MediaFS in a multimedia implementation

The MediaFS standardized interface allows higher-level multimedia applications, such as the MME, to use POSIX functions related to file and directory operations to access audio and video content along with associated metadata on media devices and mediastores.

To add a new device to a multimedia environment that uses MediaFS, all you need to do is:

- create a MediaFS implementation to represent the device according to the MediaFS requirements
- make adjustments to the client application (the MME or the HMI, or both) to ensure that they are aware of and able to handle new situations that might arise due to the presence of the new device

To create a MediaFS implementation you can use **`io-fs`**, a resource manger, or some other component as you require.

Chapter 2

MediaFS Structure

In this chapter. . .

This chapter describes:

- Filesystem location
- The MediaFS filesystem structure
- Required POSIX function support

Filesystem location

When a MediaFS implementation learns of a new media device, it registers a path with the path space manager to the location of the MediaFS filesystem; this path is called the *mountpoint*. The MediaFS implementation then:

- creates the MediaFS standarized filesystem structure for the device
- creates a filesystem representing the device under **`/fs`**
- makes available the contents of the device as a filesystem with the root directory of the device mounted on /fs/*dev_id*, where *dev_id* is a name that indicates the type of device with a numeric suffix representing the device's instance number

The first device discovered has an instance number of 0. For example, if a device is an iPod it is mounted as **`/fs/ipod0`**; while a PFS/MTP device is mounted as **`/fs/pfs0`**.

Multiple instances of a device are identified by their numeric suffixes. Thus, for example two iPods, one PFS device, and one UPnP device would be mounted as follows:

```
/fs/ipod0
/fs/ipod1
/fs/pfs0
/fs/upnp0
```

For more information about how to represent a media device to MediaFS, see the chapter MediaFS Entities.

For more information about resource managers, see the *Writing a Resource Manager* in the QNX Neutrino documentation set.

The MediaFS filesystem structure

Located under the MediaFS mountpoint, the **`.FS_info.`** directory is the MediaFS standardized structure of files and directories that contain the control and state information of a media device. Every device instance has its own MediaFS filesystem structure.

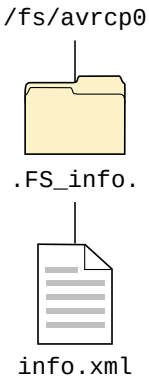
The basic MediaFS filesystem structure is as follows:

Directories
Files
mountpoint

mountpoint`/.FS_info.`
mountpoint`/.FS_info./info.xml`
mountpoint`/.FS_info./dev`
mountpoint`/.FS_info./current`
mountpoint`/.FS_info./control`
mountpoint`/.FS_info./playback`

Media device controllers must populate the the device-specific files in the `.FS.info.` directory, according to the specifications presented in the chapter MediaFS Entities.

The figure below shows a MediaFS hierarchy with a Bluetooth device.



The MediaFS module hierarchy with a Bluetooth device

Files and directories outside the `.FS_info.` directory are device dependent and, therefore, do not have a standardized, defined structure in MediaFS. See "Directories and files outside the `.FS_info.` directory" in the chapter MediaFS Entities.

Required POSIX function support

All files and folders in the MediaFS representation adhere to the POSIX standard, and the following POSIX functions must be supported on all directories and files in the MediaFS representation:

- *close()* — close a file
- *closedir()* — close a directory
- *devctl()* — control a device
- *dircntl()* — control an open directory
- *fstat()* — get file information, given a file description
- *open()* — open a file
- *opendir()* — open a directory
- *readdir()* — read a directory entry

- *stat()* — get information about a file or directory, given a path

Directories and files can be identified by using the standard POSIX *stat()* function, and the S_ISDIR and S_ISREG macros on the returned **`stat`** structure.

For speed optimizations, MediaFS should support the ability to retrieve extra *stat()* information as part of the *readdir()* operation, if the D_FLAG_STAT flag is set.

For more information about these functions and data structures, see the *QNX Neutrino Library Reference*.

Chapter 3

MediaFS Entities

In this chapter. . .

This chapter describes:

- The **.FS_info.** directory and its contents
- Files outside the **.FS_info.** directory
- Playlist files and directories

The .FS_info. directory and its contents

When a MediaFS implementation learns of a device, it creates a **.FS_info.** directory for the media device with device-specific playback and metadata interface items. The table below lists these items:

Item	Type	Required?	Description
info.xml	file	Yes	XML file with device-specific information.
control	file	Optional	File in which device-specific playback actions are issued to a media device.
dev	symbolic link	Optional	Symbolic link to device identified by the **<uuid>** element in the **info.xml** file.
current	symbolic link	Optional	Symbolic link pointing to the currently playing file in MediaFS.
playback	directory	Optional	Directory with symbolic links, listed in the same order as the media device will complete playback of the files.

See the sections below for complete descriptions of the **.FS_info.** items.

The info.xml file

The **info.xml** device information file is an XML version 1.0 file that contains device-specific information. The MediaFS implementation creates this file when it creates the **.FS_info.** directory for a media device, placing it in the root directory for the device as **.FS_info./info.xml**. This file is static and persists for the lifetime of the MediaFS instance that created it.

To the client application, the **info.xml** file is a *read only* file. Client applications can *not* write to this file.

When it creates the **info.xml** file, MediaFS does *not* populate it with device information. To enable MediaFS to present to higher level software layers a standard interface to all media devices, device controllers must populate the **info.xml** file for each device with XML-formatted, device-specific information. This XML-formatted information can be used by higher-level software, such as the MME, and may also be useful for human viewing.

The table below lists the basic elements of an **info.xml** file:

XML Key	Required?	Description
<media>	Yes	Root XML key for the media
<media>/**<device>**	Yes	A name used to indicate to upper layer components the device below the MediaFS.
<media>/**<serial>**	Yes	Device serial number
<media>/**<model>**/**<*>**	Optional	Device model information
<media>/**<protocol>**/**<*>**	Optional	Device protocol information
<media>/**<swversion>**	Optional	Device software revision
<uuid>	Optional	A unique identifier for the media device; upper layer components, such as the MME, must be able to use this key to associate the media device with its settings and remember these settings. A **<uuid>** number must be static, and unique to a media device.

Example `info.xml` file

The example below presents the minimum required content of a **info.xml** file:

```
<?xml version="1.0" standalone="yes"?>
<uuid>unique_media_identifier</uuid>
<media>
<device>devicename</device>
<serial>8N838BUH2C7</serial>
</media>
```

Mediastore changer devices, such as CD or DVD changers, require different elements in their **info.xml** file. For more information, see "The **info.xml** file for mediastore changers" in the chapter Working with Media Changers.

The `control` file

The **control** file in the **.FS_info.** directory is the file where device-specific playback actions are issued to a media device. This MediaFS control file is a file interface that supports the following I/O capabilities:

- Accept playback and state information device control messages.
- Set a state via device control messages.
- Provide asynchronous change notifications via out-of-band messaging.
- Get events from the MediaFS event queue; see also the DCMD_MEDIA_READ_EVENTS device control message, and the chapter MediaFS Events.

The MediaFS control file provides state information and metadata for the current device. That is:

- state and metadata device control messages issued on the control file return information about the device at the time of the execution
- metadata obtained from the control filet is the metadata for the currently active (playing) track

The **control** file is required for devices, such as iPods, that use serial ("two-wire") connections. It is not required for devices, such as PFS devices and certain iPods, that use USB ("one-wire") connections.

Conditions for sending a notification

If the state or the metadata of the currently active device changes, the MediaFS control file should send a notification via an out-of-band message to all registered listeners, such as, for example, the MME.

The MediaFS control file sends a notification if *any* of the following conditions is met:

- Any data that that will be returned in the **_media_playback_status** structure has changed.
- An event has been added to the MediaFS event queue.
- The **current** file has been updated.
- The content of the **playback** directory has changed.
- The device playback speed or state has changed.

To receive asynchronous notifications, a client application must use the QNX *io_notify()* function to register for these notifications.

The *dev* symbolic link

The *dev* element in the **.FS_info.** directory is a symbolic link to the media device identified by the **<uuid>** element. It should be an entry in the **/dev** directory and provide access directly to the media device.

Devices such as an HTTP client driver may not have a *dev* entry.

The *current* symbolic link

The *current* symbolic link is optional. If it is present, this symbolic link is a relative path from the **.FS_info.** directory which, when it is resolved, points to the currently active file. This active file is in the MediaFS file system.

If the **playback** directory is present, the symbolic link points to an entry in this directory. If no file on the media device is currently active, the *current* symbolic link is removed or deleted.

Metadata retrieval commands issued on the *current* symbolic link return the specified metadata for the currently playing file.

The MediaFS control file sends a notification to registered clients via an out-of-band message whenever the *current* symbolic link is updated.

A system must support the POSIX *readlink()* function in order to resolve the *current* symbolic link.

The playback directory

The **playback** directory is optional. If it is present, this directory contains symbolic links to files that the media device will play. When it writes these symbolic links to the **playback** directory, the device controller should organize them in the same order as the media device will play the files referenced by the links.

Whenever possible, symbolic links in the **playback** directory should point to the files that they represent in the main media filesystem. However, for some operational modes on some devices, the device may not be able to guarantee the accuracy of these pointers. Client applications should, therefore, treat the links as hints and not as guarantees of a file's location in the main media filesystem.

Metadata retrieval

Client applications can use metadata retrieval messages to execute metadata extraction calls against the files listed in the **playback** directory. If the links point into the MediaFS hierarchy, the results of a call to one of these symbolic links is the same as the result of a call to retrieve metadata directly from a file in MediaFS.

For more information about metadata retrieval messages, see the chapter Metadata Messages.

Changes to the playback directory

When the content of the **playback** directory changes (because, for example, the client has selected a play operation against a new set of media files), the control file sends a notification of the change to all applications registered for out-of-band messages on the **control** file.

Directories and files outside the .FS_info. directory

The behavior of MediaFS entities located outside the **.FS_info.** directory structure varies according to the capabilities and behaviors of each underlying media device. However, MediaFS maintains some behaviors throughout for these directories and files, as described in this section.

Directory behavior

MediaFS directories outside the **.FS_info.** directory represent the data hierarchy of the target device. For example, directories on an Apple iPod could be represented using the following structure:

```
/ipod0
/ipod0/.FS_info.
/ipod0/Music
/ipod0/Music/Artists
/ipod0/Music/Songs
...
```

Directory characteristics

To be usable by MediaFS, a directory outside the **.FS_info.** directory structure must have the following characteristics:

- The directory must be of the type S_ISDIR.
- The directory must support the DCMD_FSYS_DIR_NFILES command message, to indicate the number of files present in the directory.
- All directory attributes (name, size, etc.) must adhere to the QNX filesystem specifications and POSIX specifications.

Additionally, directories outside the **.FS_info.** directory structure may need to accept the DCMD_MEDIA_PLAY command, if the associated media device supports playback of all items in a directory; that is, if the device supports using a directory as a playlist.

File behavior

MediaFS files outside the **.FS_info.** directory represent files or tracks that can used with the media device.

The POSIX filesystem representation for these files is free-form, with the following restrictions:

- Files must be of the type S_ISREG.
- Files should use a file extension to aid in file type detection.
- All file attributes (name, size, etc.) must adhere to QNX filesystem specifications and POSIX specifications.

Supported device control messages

MediaFS files outside the **.FS_info.** directory structure must support state change and metadata query device control messages.

Messages issued directly to a file outside this directory structure must apply to the specified file. For example, the device control message DCMD_MEDIA_SONG issued

directly to a file outside the **.FS_info.** directory structure returns the song title for that file, *not* the song title of the currently playing file.

If a device control message cannot be completed due to media device limitations, the call that issues the control message returns an ENOTSUP error.

For a complete list of device control messages used with MediaFS and descriptions of these messages, see the chapters Playback Messages and Metadata Messages.

Playlist files and directories

A playlist can be either of:

- a standardized playlist file, such as an M3U or PLS file, stored in the MediaFS hierarchy; entries in the these playlist files must be filesystem paths pointing to entries in MediaFS
- a MediaFS playlist, which is a collection of files inside a directory

Playlist support is subject to upper layer component (MME) support.

MediaFS playlists

Note the following about MediaFS playlists:

- A MediaFS playlist directory can be present only *outside* of the **.FS_info.** directory.
- For a directory to be identified as a MediaFS playlist, the directory must have the *others* execute bit cleared in the *st_mode* member of its **stat** structure.

Indentifying a MediaFS playlist

The code snippet below shows how to determine if a directory is a MediaFS playlist:

```
stat(path, &statbuf);
if ((S_ISDIR ! S_IXOTH) & statbuf.st_mode)==S_ISDIR) {
    //This is a MediaFS playlist
}
```

For more information about the **stat** structure, see *stat(), stat64()* in the the *QNX Neutrino Library Reference*.

Chapter 4

Media Changers

In this chapter. . .

Media changer devices — essentially CD and DVD changers — differ from other media devices because they contain and change *removable mediastores*.

A removable mediastore is a physical storage medium, such as a CD or DVD, with one of more media files that can be synchronized and played. Media changer devices can load and unload these mediastores as required, changing their states from "unavailable" to "available" to "active".

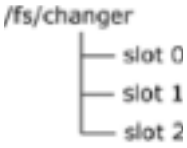
This chapter describes how these devices and their mediastores can be represented to MediaFS, and how state changes on these devices should be communicated to MediaFS:

- Representing media changers and mediastores
- Informing MediaFS of state changes

Representing media changers and mediastores

MediaFS uses extensions to offer a common representation of devices with multiple mediastores — devices such as CD and DVD changers. It represents a media changer device as a single *changer* container. This changer container includes multiple *slot* items. Each slot represents one mediastore (such as a CD or a DVD), and is described by a separate MediaFS instance.

For example, the following illustration represents the hierarchy of one media changer device with three slots, each slot described by a MediaFS instance:



MediaFS changer device and mediastore hierarchy representation

MediaFS instances for slots

A slot represents a single mediastore. Each slot has a MediaFS instance, which adheres to the MediaFS specifications for media device representation. That is, each slot has a MediaFS instance with its own **`info.xml`** file, **`control`** file, optional **`playback`** directory, *dev* symbolic link, and optional *current* symbolic link.

The `info.xml` file for mediastore changers

MediaFS **`info.xml`** files for changer devices and slots differ from **`info.xml`** files for other media devices in order to accurately represent the devices as containers for the mediastores, and the mediastores as dependent on a device. That is, the **`info.xml`** files for changer devices and slots must indicate that the changer device can hold one, many or no mediastores, and that these mediastores can only be accessed inside a changer.

The table below lists the elements required in an **info.xml** file use to describe media changer devices and slots:

XML Key	Required?	Description
<media>	Yes	Root XML key for the media
<media>/<driver>	Yes	Description of the device as a MediaFS changer device
<media>/<name>	Yes	Name of the mediastore
<media>/<serial>	Yes	Device serial number
<media>/<slot>	Yes	Slot number for the mediastore
<media>/<type>	Yes	Description of the mediastore type

Required XML keys

The keys listed below are required in the **info.xml** file for a media changer device slot:

<media>/<driver>

A user-defined name for the device; for slots this value *must* be set to **mediafs-changer**.

<media>/<name>

The mediastore name that can be passed to the upper software layers (such as the MME and an HMI) for display to the user. In most cases this name is the volume name of the mediastore.

<media>/<serial>

A unique identifier for the mediastore represented by the slot. It must be set to a value , such as the freeDB hash, that uniquely identifies the mediastore.

<media>/<slot>

The slot position of the mediastore in the changer device. The value of this key must be the same as the offset returned by a call to the device with the DCMD_CAM_MECHANISM_STATUS control message. This offset (and therefore the value of the **<slot>** key) is a single-digit string representing the slot with the mediastore.

<media>/<type>

A predefined text value identifying the kind of mediastore present in the media changer device slot. Permitted values are:

- FS — basic filesystem
- AUDIOCD — CDDA disc
- DVDVIDEO — DVD-video disc

- DVDAUDIO — DVD-audio disc
- VCD — Video CD disc
- UNKNOWN — unknown mediastore type

Example `info.xml` file

The example below shows an **`info.xml`** file for a MediaFS slot representing a mediastore changer device:

```
<?xml version="1.0" standalone="yes"?>
<info>
   <media>
        <device>mediafs-changer</device>
        <slot>1</slot>
        <serial>280a1752</serial>
        <name>MIXED</name>
        <type>FS</type>
   </media>
   <device>
        <driver>mediafs-changer</driver>
        <catagory>media</catagory>
   </device>
</info>
```

For more information about the **`info.xml`** file, see "The **`info.xml`** file" in the chapter MediaFS Entities.

Informing MediaFS of state changes

MediaFS expects a slot to have one of the following states:

- unavailable — the slot is not represented in MediaFS
- available — the slot is represented in MediaFS, inside the changer container
- active — the slot is available *and* a mediastore that can be synchronized and played is physically present

For example, if a CD changer has six possible mediastore locations, it can be represented by a changer with any one of slots 0 to 5. If a mediastore is loaded into slot 0, MediaFS represents it as shown in the figure below:

MediaFS changer representation of a mediastore in slot 0

If the mediastore is ejected from the changer, MediaFS removes its slot representation:

/fs/changer

MediaFS changer representation of device with no available mediastores

For files on a mediastore to be synchronized or played, the slot representing the mediastore in MediaFS must by marked *active* as well as available. That is, the device controller must use the slot's **control** file to inform MediaFS not only that the slot is present, but that a readable disc is physically loaded in the changer and is ready to be read. Thus, only one changer slot can be active at any one time.

Changer states

To get the current changer state from MediaFs, the client application or the device controller must issue the DCMD_CAM_CDROM_MECHANISM_STATUS device control message to each changer slot's **control** file, as appropriate.

DCMD_CAM_CDROM_MECHANISM_STATUS is a standard control defined in the **sys/cdrom.h** header file. The example below shows one way to implement the DCMD_CAM_CDROM_MECHANISM_STATUS device control command:

```
#define CDROM_MSH_CHANGER_SET_CURRENT_SLOT( cdrom_status, slot )\
    cdrom_status.mech_state         &= ~0x07         ; \
    cdrom_status.changer_state_slot &= ~0x1F         ; \
    cdrom_status.mech_state         |= (slot >> 5)   ; \
    cdrom_status.changer_state_slot |= (slot & 0x1F) ;

struct _cdrom_mechanism_status  cdrom_status ;
struct _cdrom_exchange          cdrom_exchange ;

switch(changer.status)
{
case STATUS_EMPTY:
    cdrom_status.hdr.mech_state = CDROM_MSH_MECHANISM_IDLE ;
    break;
case STATUS_RETRACT:
case STATUS_LOAD:
    cdrom_status.hdr.changer_state_slot = CDROM_MSH_CHANGER_LOADING ;
    break;
case STATUS_UNLOAD:
    cdrom_status.hdr.changer_state_slot = CDROM_MSH_CHANGER_UNLOADING ;
    break;
default:
    cdrom_status.hdr.changer_state_slot = CDROM_MSH_CHANGER_READY ;
    break;
}

cdrom_status.hdr.num_slots_avail = changer.num_slots ;
cdrom_status.hdr.slot_table_len  = changer.num_slots ;

for( index=0; index < changer.num_slots; index++ ) {
    if(changer.slotInfo[index].status == DISCIN) {
```

```
            cdrom_status.str[index].flags |= CDROM_STR_DISC_PRESENT;
        }
    }

    CDROM_MSH_CHANGER_SET_CURRENT_SLOT( cdrom_status.hdr,
                                        changer.active_slot);
```

Chapter 5

Managing Playback

In this chapter. . .

Playback on media devices may be initiated or changed by:

- a user; that is, a high-level application, such as the MME
- a media device, such as an iPod

This chapter presents the control message sequences and the settings that a device controller may need to support to monitor and manage media playback through *devctl()* calls to MediaFS entities. It contains the following sections:

- Requested playback control sequences
- Managing autonomous playback state changes

For a list of MediaFS control messages, see the chapters Playback Messages and Metadata Messages.

Requested playback control sequences

A media device controller using MediaFS should support client applications issuing commands to MediaFS entities to start playback of:

- a media file (track), at the start of the track or at an offset, if the media device supports playback from an offset
- a directory
- a media device, if the media device supports this action

This section presents the control message sequences and settings required to effect a playback state change through a *devctl()* call to a MediaFS entity. It contains:

- Start playback — file or directory
- Start playback — media device
- Fast forward and reverse
- Pause and resume playback

Start playback — file or directory

To start playback for a specific file or directory, the client application should issue, as required, either the DCMD_MEDIA_PLAY or the DCMD_MEDIA_PLAY_AT device control message to the MediaFS file or directory to play.

If the file or directory is valid for the media device, the device controller must perform the following operations, in sequence:

1. Receive the device control message, and validate playback.
2. Start playback of the requested track on the media device.
3. Update the following **`_media_playback`** structure members:

- *count* — set to the number of tracks that will be played
- *index* — set to the index of the requested track
- *state* — set to PLAYBACK_STATE_PLAY
- *flags* — if the media device supports this feature, set to PLAYBACK_FLAG_SPEED_EXACT *only*; if the device does not support this feature, set to 0 (zero)
- *metaseq* — set to 0 (zero)
- *length* — set to the length of the track, or to 0 (zero) if the track length is not available
- *elapsed* — set to 0 (zero) if the DCMD_MEDIA_PLAY message was issued, or to the track start offset, in seconds, if the DCMD_MEDIA_PLAY_AT message was issued and is supported
- *speed* — if the the *flags* member is set to PLAYBACK_FLAG_SPEED_EXACT, set to 1 (one) *only*; no other value is permitted

4. Update the MediaFS *current* symbolic link (if it is present) to point to the requested media file.
5. Send an out-of-band notification on the **`control`** file.
6. If all operations are successful, reply to the device control message with EOK.

Start playback — media device

Some media devices support playback of the entire device, starting with the first track in the device, a random track, or at the point where playback was previously stopped. To start or resume playback of a device, a client application should issue the DCMD_MEDIA_PLAY device control message to the MediaFS **`control`** file.

If this action is valid for the current media device, the device controller must perform the following operations, in sequence:

1. Receive the device control message, and validate playback.
2. Start playback of the media device.
3. Update the following **`_media_playback`** structure members:
   - *count* — set to either 1 (one) if only one track will be played, or to the number of tracks that will be played
   - *index* — set to the currently playing track if the device provides this information immediately, or to 0 (zero) if the information is not provided at this time
   - *state* — set to PLAYBACK_STATE_PLAY
   - *flags* — if the media device supports this feature, set to PLAYBACK_FLAG_SPEED_EXACT *only*; no other value is permitted

- *metaseq* — set to 0 (zero)
- *length* — set to the length of the track, or 0 (zero) if the length is unavailable
- *elapsed* — set to 0 (zero)
- *speed* — set to 1 (one), *only* if the PLAYBACK_FLAG_SPEED_EXACT flag is set; no other value is permitted

4. Update the *current* symbolic link to point to the requested media file. If the currently playing file is not known at this time, clear the symbolic link.
5. Send an out-of-band notification on the **`control`** file.
6. If all operations are successful, reply to the device control message with EOK.

Track information updates

If a device autonomously indicates the playing track *after* MediaFS has replied to the DCMD_MEDIA_PLAY device control message that started playback, the device controller must perform the following operations, in sequence:

1. Update the following **`_media_playback`** structure members:
   - *count* — set to either 1 (one) if only one track will be played, or to the number of tracks that will be played
   - *index* — set to the currently playing track
   - *metaseq* — increment by 1 (one), if metadata is now available
   - *length* — set to the length of the track, or 0 (zero) if the length is unavailable
   - *elapsed* — set to the current track time received from the media device
2. Send an out-of-band notification on the **`control`** file.

Fast forward and reverse

The playback speed or direction of a media device represented through MediaFS can be changed only while the device is in the playing state. To change the playback speed or direction, or both, the client application should issue the appropriate messages to the the MediaFS **`control`** file.

If the action is valid for the current media device, the device controller must perform the following operations, in sequence:

1. Receive the device control message, and validate playback.
2. Change the playback speed on the media device, as requested.
3. Update the following **`_media_playback`** structure members:
   - *state* — set to PLAYBACK_STATE_PLAY (1)
   - *flags* — set to PLAYBACK_FLAG_FASTFWD or PLAYBACK_FLAG_FASTRWD, and set to PLAYBACK_FLAG_SPEED_EXACT if the device supports this feature

- *speed* — set to the playback speed, *only* if the PLAYBACK_FLAG_SPEED_EXACT flag is set

4. Send an out-of-band notification on the **`control`** file.
5. If all operations are successful, reply to the device control message with EOK.

For more information about fast forward and reverse control messages, see the chapter Playback and Status Messages.

Pause and resume playback

The client application should pause and resume playback on MediaFS devices by issuing messages to the MediaFS **`control`** file. If these actions are valid for the current media device, the device control should apply them to the device.

Pause playback

Playback can only be paused while a device is in the playing state. To pause playback, a client application should issue the DCMD_MEDIA_PAUSE message to the MediaFS **`control`** file. If this action is valid for the current media device, the device controller must perform the following operations, in sequence:

1. Receive the device control message, and validate playback.
2. Pause playback on the media device.
3. Update the following **`_media_playback`** structure members:
   - *state* — set to PLAYBACK_STATE_PAUSE
   - *speed* — set to 0 (zero), *only* if the PLAYBACK_FLAG_SPEED_EXACT flag is set
4. Send an out-of-band notification on the **`control`** file.
5. If all operations are successful, reply to the device control message with EOK.

Resume playback

Playback can only be resumed while the device is in the paused state. To resume playback, the client application should issue the DCMD_MEDIA_RESUME message to the MediaFS **`control`** file. If this action is valid for the current media device, the device controller must perform the following operations, in sequence:

1. Receive the device control message, and validate playback.
2. Resume playback paused on the media device.
3. Update the following **`_media_playback`** structure members:
   - *state* — set to PLAYBACK_STATE_PLAY
   - *speed* — set to the device speed, *only* if the PLAYBACK_FLAG_SPEED_EXACT flag is set

4. Send an out-of-band notification on the **`control`** file.
5. If all operations are successful, reply to the device control message with EOK.

Managing autonomous playback state changes

During playback, a media device may change playback or metadata states autonomously, independently of any user request. This section describes the actions that a device controller must perform when it encounters a device-initiated state change:

- Track change
- Playback state change
- Metadata update

Track change

If a media device autonomously changes tracks, the device controller must perform the following operations, in sequence:

1. Update the following **`_media_playback`** structure members:
   - *count* — set to the number of tracks that will be played
   - *index* — set to the new currently playing track
   - *metaseq* — set to 0
   - *length* — set to the length of the track, or 0 (zero) if the length is unavailable
   - *elapsed* — set to 0 (zero)
2. Update the *current* symbolic link to point to the new currently playing MediaFS file.
3. Send an out-of-band notification on the **`control`** file.

Playback state change

Some media devices may autonomously pause, stop, or resume playback. If these state changes occur on a media device, the device controller must perform the following operations, in sequence:

1. Update the following **`_media_playback`** structure members:
   - *state* — set to the new playback state
   - *speed* — set to the device speed, *only* if the PLAYBACK_FLAG_SPEED_EXACT flag is set
2. Send an out-of-band notification on the **`control`** file.

Metadata update

If a media device supports asynchronous metadata updates, it may update the metadata for the current playing track. If an update of this type occurs, the device controller must perform the following operations, in sequence:

1. Update the following **`_media_playback`** structure member:
   - *metaseq* — increment by 1 (one)
2. Send an out-of-band notification on the **`control`** file.

Subsequent client application requests for metadata (made through a device control message to a MediaFS entity) will retrieve the new metadata that was received from the media device.

Chapter 6

Device Messages

In this chapter. . .

This chapter describes the MediaFS device control messages and how to use them. It contains the following sections:

- Using device control messages
- Device configuration messages
- iPod, UPnP device, and streaming messages

- The MediaFS device control messages, constants and data structures are defined in the header file **`io-fs/lib/public/sys/dcmd_media.h`**.
- If a buffer is required for a command message, this documentation includes a buffer description with the message description; the buffer description follows the template: “Buffer: *description*”. If the control message does not require a buffer, then no buffer description is presented with the message description.
- For information about how to use MediaFS device control messages, see “Using device control messages” below.
- For information about the messages used to retrieve album art, see “Album art retrieval messages” in the chapter Getting Album Art.

Using device control messages

MediaFS device control messages can be applied to open files and directories in the MediaFS filesystem to:

- query media devices for their states and playback information
- initiate actions against media files, such as start, pause and stop playback, skip to the next or previous file, or change random and repeat mode settings
- retrieve file metadata
- extract album art and other images

Control messages are applied by calls to the *devctl()* function. When a control message is applied to a MediaFS entity, the filesystem routes the message to the appropriate device driver. The device driver must:

- apply the requested action to the underlying media device
- return to the calling application the the result of the action

All state modification control messages must be synchronous. A requested action must either complete or fail before returning. For example, if the state modifier DCMD_MEDIA_PLAY message is issued, upon return of the *devctl()* call, the underlying device must be in a playing state, or have returned a POSIX error indicating why the command failed.

As with other directories and files, an application must open a MediaFS directory or file with, respectively, the *opendir()* and the *open()* functions before it can use the *devctl()* function to issue control messages to them.

To pass data to and receive data from media devices, a client application should use the *devctl()* function's *dev_data_ptr* and *dev_info_ptr* arguments to point to the appropriate **_media_*** data structures. These **_media_*** structures are described in the chapter Playback Structures and Constants.

For a list of POSIX functions that MediaFS supports, see "Required POSIX function support" in the chapter MediaFS Structure. For more information about the functions, such as *open()* and *devctl()*, used to control devices, see the *QNX Neutrino Library Reference*.

Device configuration messages

This section describes the device messages defined to get and set device configurations. These message are:

- DCMD_MEDIA_GET_XML
- DCMD_MEDIA_SET_XML

DCMD_MEDIA_GET_XML

DCMD_MEDIA_GET_XML returns an XML configuration string (UTF-8) with the device configuration information. See also the chapter MediaFS Entities.

Buffer: **char[1]**

DCMD_MEDIA_SET_XML

DCMD_MEDIA_SET_XML expects a buffer containing a terminated xpath string followed by a terminated value string; that is, the element or attribute to modify, and its new value.

Buffer: **char[1]**

DCMD_MEDIA_GET_XML and DCMD_MEDIA_SET_XML use a common configuration layout that becomes specific for each device. For example:

```
<device api_version="1">
    <media>
        <interface type="usb" ... />
        <DeviceSpecificString>
        .... Device specific settings
        </DeviceSpecificString>
    </media>
</device>
```

See also the chapter MediaFS Entities.

iPod, UPnP device and streaming messages

This section describes the device control messages defined to obtain information from and manage iPod and UPnP devices, and media streams. These message are:

- Common messages
- iPod device messages
- UPnP device messages
- Media stream messages

Most devices do not support the full set of control messages. If a device does not support a requested action, the device controller must return the error code ENOTSUP (command invalid for this device).

Common messages

This section describes the device control messages that can be used to obtain information from and manage iPod devices, UPnP devices, DRM, and media streams. These messages are:

- DCMD_MEDIA_CONFIG
- DCMD_MEDIA_GET_DEVINFO

DCMD_MEDIA_CONFIG

DCMD_MEDIA_CONFIG issues a configuration setting to a media device.

Buffer: **`char[1]`**

DCMD_MEDIA_GET_DEVINFO

DCMD_MEDIA_GET_DEVINFO requests information about a media device.

Buffer: **`char[8*1024]`**

iPod device messages

This section describes the device control messages defined to obtain information from and manage iPod devices. These message are:

- DCMD_MEDIA_IPOD_DAUDIO
- DCMD_MEDIA_IPOD_CAP
- DCMD_MEDIA_IPOD_TAG

DCMD_MEDIA_IPOD_DAUDIO

DCMD_MEDIA_IPOD_DAUDIO is used to retrieve iPod audio settings from an iPod device. This information is carried in the **`_media_ipod_daudio`** data structure

Buffer: **`struct _media_ipod_daudio`**

DCMD_MEDIA_IPOD_CAP

DCMD_MEDIA_IPOD_CAP retrieves capabilities information from an iPod device.

Buffer: **`char[1]`**

DCMD_MEDIA_IPOD_TAG

DCMD_MEDIA_IPOD_TAG is used to add an iTunes tag to a file on an iPod device.

Buffer: **`uint8_t[1]`**

UPnP device messages

This section describes the device control messages defined to obtain information from and manage UPnP devices. These message are:

- DCMD_MEDIA_UPNP_CDS_BROWSE

DCMD_MEDIA_UPNP_CDS_BROWSE

DCMD_MEDIA_UPNP_CDS_BROWSE browses a mediastore on a device that uses the UPnP protocol.

Buffer: **`char[8]`**

Media stream messages

This section describes the device control messages defined to obtain information from and manage media streams. These message are:

- DCMD_MEDIA_CLOSE_STREAM

- DCMD_MEDIA_INFO_STREAM
- DCMD_MEDIA_OPEN_STREAM
- DCMD_MEDIA_READ_STREAM
- DCMD_MEDIA_SET_STREAM

DCMD_MEDIA_CLOSE_STREAM

DCMD_MEDIA_CLOSE_STREAM closes a media stream.

DCMD_MEDIA_INFO_STREAM

DCMD_MEDIA_INFO_STREAM retrieves information about a media stream. This information must be placed in a **`_media_stream_info`** structure.

Buffer: **`struct _media_stream`**

DCMD_MEDIA_OPEN_STREAM

DCMD_MEDIA_OPEN_STREAM opens a media stream.

DCMD_MEDIA_READ_STREAM

DCMD_MEDIA_READ_STREAM reads a media stream into a buffer. Before a stream can be read, it must be opened with a DCMD_MEDIA_OPEN_STREAM message and set with a DCMD_MEDIA_SET_STREAM message.

Buffer: **`char[16*1024-1]`**

DCMD_MEDIA_SET_STREAM

DCMD_MEDIA_SET_STREAM sets the media stream that will be read by calls to *devctl()* with the DCMD_MEDIA_READ_STREAM message.

Buffer: **`unsigned int`**

Chapter 7

Playback Messages

This chapter describes the MediaFS device control messages defined to retrieve state information from and control playback on a media device accessed and managed through MediaFS, or that are outside MediaFS. These message are:

- DCMD_MEDIA_ACCESS_TYPE
- DCMD_MEDIA_FASTFWD
- DCMD_MEDIA_FASTRWD
- DCMD_MEDIA_GET_REPEAT
- DCMD_MEDIA_GET_SHUFFLE
- DCMD_MEDIA_GET_STATE
- DCMD_MEDIA_NEXT_CHAP
- DCMD_MEDIA_NEXT_TRACK
- DCMD_MEDIA_PAUSE
- DCMD_MEDIA_PLAY
- DCMD_MEDIA_PLAY_AT
- DCMD_MEDIA_PLAYBACK_INFO
- DCMD_MEDIA_PLAYBACK_STATUS
- DCMD_MEDIA_PREV_CHAP
- DCMD_MEDIA_PREV_TRACK
- DCMD_MEDIA_RESUME
- DCMD_MEDIA_SEEK_CHAP
- DCMD_MEDIA_SET_REPEAT
- DCMD_MEDIA_SET_SHUFFLE
- DCMD_MEDIA_SET_STATE

Playback control and device status messages can be issued to the MediaFS **`control`** file *only*.

The exceptions to this rule are:

- the DCMD_MEDIA_PLAY message, which can be issued to:
  - the MediaFS **`control`** file
  - a MediaFS file or directory
  - any other file

  - a directory, if the device supports directory playback
- the DCMD_MEDIA_PLAY_AT message, which can be issued to:
  - a MediaFS file

- Most devices do not support the full set of control messages. If a message is not supported by the media device the requested action, the device controller must return the error code ENOTSUP (command invalid for this device).
- All state modification control messages must be synchronous; the requested action must either complete or fail before returning.

  For example, if the state modifier DCMD_MEDIA_PLAY is issued, upon return of the *devctl()* call, the underlying device must be in a playing state, or have returned a POSIX error indicating why the command failed.

DCMD_MEDIA_ACCESS_TYPE

DCMD_MEDIA_ACCESS_TYPE retrieves information about how a file is to be accessed. The status values may indicate that the file is known to be either DRM protected or *not* DRM protected, as well as whether a POSIX *read()* function can be used to access the file's media content.

This command does not use a data transfer buffer. If the *devctl()* status is EOK, the integer return value (obtained using the last parameter of the *devctl()* function) may contain a combination of the following values:

Type	Value	Description
ACCESS_TYPE_DRM_PROTECTED	1	The file is known to be DRM protected. May *not* be combined with ACCESS_TYPE_DRM_UNPROTECTED.
ACCESS_TYPE_DRM_UNPROTECTED	2	The file is known to not be DRM protected. May *not* be combined with ACCESS_TYPE_DRM_PROTECTED.
ACCESS_TYPE_READ_SUPPORTED	4	The file can be read using POSIX read functions (the file is not Zune). May be combined with ACCESS_TYPE_DRM_UNPROTECTED *or* with ACCESS_TYPE_DRM_PROTECTED.

DCMD_MEDIA_FASTFWD

DCMD_MEDIA_FASTFWD instructs the media device to go to the fast forward speed specified by the *rate* member of the **_media_speed** structure. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

Buffer: **struct _media_speed**

DCMD_MEDIA_FASTRWD

DCMD_MEDIA_FASTRWD instructs the media device to go to the fast reverse speed specified by the **_media_speed** data structure's *rate* member. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

Buffer: **struct _media_speed**

DCMD_MEDIA_GET_REPEAT

DCMD_MEDIA_GET_REPEAT queries the media device for its current repeat playback mode. Defined repeat modes are:

- REPEAT_OFF
- REPEAT_ONE_TRACK
- REPEAT_ALL_TRACKS
- REPEAT_FOLDER
- REPEAT_SUBFOLDER

On success, the call must return the current device repeat mode, in the **_media_settings** data structure's *value* member.

Buffer: **struct _media_settings**

DCMD_MEDIA_GET_SHUFFLE

DCMD_MEDIA_GET_SHUFFLE queries the media device for its current random playback mode. Defined random modes are:

- SHUFFLE_OFF
- SHUFFLE_TRACKS
- SHUFFLE_ALBUMS
- SHUFFLE_FOLDER
- SHUFFLE_SUBFOLDER

On success, the call must return the current device random mode, in the **_media_settings** data structure's *value* member.

Buffer: **struct _media_settings**

DCMD_MEDIA_GET_STATE

DCMD_MEDIA_GET_STATE queries the media device for its current settings and returns the data in the **_media_settings** data structure. This data can be used at a later time to restore playback to the state at the time of the query.

Buffer: **uint8_t[1]**

DCMD_MEDIA_NEXT_CHAP

DCMD_MEDIA_NEXT_CHAP instructs the media device to skip forward to the next chapter in a video. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

DCMD_MEDIA_NEXT_TRACK

DCMD_MEDIA_NEXT_TRACK instructs the media device to skip forward to the next file in its playlist. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

DCMD_MEDIA_PAUSE

DCMD_MEDIA_PAUSE instructs the media device to pause playback of the current file. Issuing this message always causes a "pause" instruction to be sent to the media device, *even when playback is already in a paused state*.

DCMD_MEDIA_PLAY

DCMD_MEDIA_PLAY directs a media device to start playback. Behavior depends on the entity to which this message is issued, as follows:

- file — start or resume playback of the current file
- directory — start or resume playback of the file in the directory, as specified by the media device
- control file — start or resume playback of a track determined by the media device

DCMD_MEDIA_PLAY_AT

DCMD_MEDIA_PLAY_AT instructs the media device to start playback at a specified time offset in a file. This play time offset is set in the **_media_play** data structure.

Buffer: **struct _media_play**

DCMD_MEDIA_PLAYBACK_INFO

DCMD_MEDIA_PLAYBACK_INFO queries the media device for its current playback information and returns the data in the **_media_playback** data structure.

All media devices must support this capability, as it is fundamental to executing playback.

Buffer: **struct _media_playback**

DCMD_MEDIA_PLAYBACK_STATUS

DCMD_MEDIA_PLAYBACK_STATUS queries the media device for its current playback status and returns the data in the **_media_playback_status** data structure.

All media devices must support this capability, as it is fundamental to executing playback.

Buffer: **struct _media_playback_status**

DCMD_MEDIA_PREV_CHAP

DCMD_MEDIA_PREV_CHAP instructs the media device to skip back to the previous chapter in a video. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

DCMD_MEDIA_PREV_TRACK

DCMD_MEDIA_PREV_TRACK instructs the media device to skip backward to the previous file in its playlist. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

DCMD_MEDIA_RESUME

DCMD_MEDIA_RESUME instructs the media device to resume the playback of the current file. Issuing this message always causes a "resume" instruction to be sent to the media device, *even when playback has already resumed*.

DCMD_MEDIA_SEEK_CHAP

DCMD_MEDIA_SEEK_CHAP instructs the media device to seek to the specified chapter in a video. Behavior when this message is issued to a media device that is not in a playing state is device dependent: the request may succeed or fail, depending on the media device's capabilities and characteristics.

Buffer: **uint32_t**

DCMD_MEDIA_SET_REPEAT

DCMD_MEDIA_SET_REPEAT sets the repeat mode on the media device. For a list of defined repeat modes, see DCMD_MEDIA_GET_REPEAT above.

Buffer: **`struct _media_settings`**

DCMD_MEDIA_SET_SHUFFLE

DCMD_MEDIA_SET_SHUFFLE sets the random (shuffle) mode on the media device, changing the playback order. For a list of defined random modes, see DCMD_MEDIA_GET_SHUFFLE above.

Buffer: **`struct _media_settings`**

DCMD_MEDIA_SET_STATE

DCMD_MEDIA_SET_STATE restores the playback settings on the media device to the values stored in the **`_media_settings`** data structure by a *devctl()* call with the DCMD_MEDIA_GET_STATE message.

Buffer: **`uint8_t[1]`**

Chapter 8

Metadata Messages

This chapter describes the MediaFS device control messages defined to obtain media file metadata from a media device accessed and managed through MediaFS, or that are outside MediaFS. These message are:

- DCMD_MEDIA_ALBUM
- DCMD_MEDIA_ARTIST
- DCMD_MEDIA_COMMENT
- DCMD_MEDIA_COMPOSER
- DCMD_MEDIA_DURATION
- DCMD_MEDIA_GENRE
- DCMD_MEDIA_NAME
- DCMD_MEDIA_PUBLISHER
- DCMD_MEDIA_RELEASE_DATE
- DCMD_MEDIA_SONG
- DCMD_MEDIA_TRACK_NUM
- DCMD_MEDIA_URL

Metadata retrieval messages to can be issued to:

- the MediaFS **`control`** file
- files entries in the MediaFS **`playback`** directory
- the **`current`** symbolic link
- any file not in the **`.FS_info.`** directory

Behavior of metadata requests

Metadata retrieved by a call to *devctl()* with a DCMD_MEDIA_* metadata retrieval message is returned as a NULL-terminated string.

Return

If the queried media device does no support the requested metadata query, the *devctl()* call with the DCMD_MEDIA_* metadata query message returns ENOTSUP.

Metadata for the currently playing file

To request metadata for the currently playing media file, use a metadata retrieval message with a call to the MediaFS **`control`** file, or to the **`current`** symbolic link.

Successful completion of a *devctl()* call with a metadata retrieval device control message to the **`control`** file or to the **`current`** symbolic link retrieves the requested metadata for the *currently playing* file.

Metadata for a specified file

To request metadata for a specific media file, use a metadata retrieval message with a call to that file.

Successful completion of a *devctl()* call with a metadata retrieval device control message to a file that is *not* the MediaFS **`control`** file or the **`current`** symbolic link retrieves the requested metadata for the *specified* file.

DCMD_MEDIA_ALBUM

DCMD_MEDIA_ALBUM queries a file for its album metadata, which the call returns in a NULL-terminated string. An empty string is valid if the album metadata is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_ARTIST

DCMD_MEDIA_ARTIST queries a file for its artist metadata, which the call returns in a NULL-terminated string. An empty string is valid if the artist metadata is not knowm.

Buffer: **`char[1]`**

DCMD_MEDIA_COMMENT

DCMD_MEDIA_COMMENT queries a file for its comment metadata, which the call returns in a NULL-terminated string. An empty string is valid if there is no track comment metadata.

Buffer: **`char[1]`**

DCMD_MEDIA_COMPOSER

DCMD_MEDIA_COMPOSER

DCMD_MEDIA_COMPOSER queries a file for its composer metadata, which the call returns in a NULL-terminated string. An empty string is valid if the composer metadata is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_DURATION

DCMD_MEDIA_DURATION

DCMD_MEDIA_DURATION queries a file for its duration, which the call returns as an unsigned integer indicating the track duration, in seconds.

Buffer: **`char[1]`**

DCMD_MEDIA_GENRE

DCMD_MEDIA_GENRE queries a file for its genre metadata, which the call returns in a NULL-terminated string. An empty string is valid if the genre metadata is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_NAME

DCMD_MEDIA_NAME queries a file for its name, which the call returns in a NULL-terminated string. An empty string is valid if the name is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_PUBLISHER

DCMD_MEDIA_PUBLISHER queries a file for its publisher metadata, which the call returns in a NULL-terminated string. An empty string is valid if the track number is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_RELEASE_DATE

DCMD_MEDIA_RELEASE_DATE queries a file for its release data metadata, which the call returns in the **`_media_date`** data structure.

Buffer: **`char[1]`**

DCMD_MEDIA_SONG

DCMD_MEDIA_SONG queries a file for the song title, which the call returns in a NULL-terminated string.

Buffer: **`char[1]`**

DCMD_MEDIA_TRACK_NUM

DCMD_MEDIA_TRACK_NUM queries a file for its track number, which the call returns in a NULL-terminated string. An empty string is valid if the track number is not known.

Buffer: **`char[1]`**

DCMD_MEDIA_URL

DCMD_MEDIA_URL gets the URL for a media file.

Buffer: **`char[1]`**

Chapter 9

Playback Structures and Constants

In this chapter. . .

Thsi chapter describes MediaFS structures and constants used for playback monitoring and control:

- Playback structures
- Playback constants
- iPod structures

Playback structures

MediaFS uses the following data structures to report and control playback information of files in the MediaFS framework:

- **_media_date**
- **_media_play**
- **_media_playback**
- **_media_playback_status**
- **_media_settings**
- **_media_speed**
- **_media_stream_info**

_media_date

```
struct _media_date {
    uint16_t year;
    uint8_t  second;
    uint8_t  minutes;
    uint8_t  hours;
    uint8_t  day;
    uint8_t  month;
    uint8_t  weekday;
    char     text[40];
}
```

The **_media_date** structure contains track date information. It is populated and returned by *devctl()* when this function successfully issues a DCMD_MEDIA_RELEASE_DATE message to a MediaFS file.

Member	Type	Description
year	**uint16_t**	The release date year (0000-9999).
second	**uint8_t**	The release date second (00-59).

continued...

Member	Type	Description
minutes	**uint8_t**	The release date minute (00-59).
hours	**uint8_t**	The release date hour (00-59).
day	**uint8_t**	The release date day (01-31).
month	**uint8_t**	The release date month (01-12).
weekday	**uint8_t**	The release date day of the week (0-6), starting with 0 for Sunday and going to 6 for Saturday.
text	**char**	A free-form, NULL text field for date information for use with devices that cannot store date specifics. Maximum length is 39 characters. If this field is used, all other fields in this structure must be set to 0 (zero).

_media_play

```
struct _media_play {
    unsigned pos;
};
```

The **_media_play** structure is used in combination with the DCMD_MEDIA_PLAY_AT command to set the starting play position. It includes at least the members described in the table below.

Member	Type	Description
pos	**unsigned**	The offset from time zero, in seconds, at which to start playback.

_media_playback

```
struct _media_playback {
    uint32_t count;
    uint32_t index;
    uint8_t  state;
    uint8_t  flags;
    uint16_t metaseq;
    uint32_t length;
    uint32_t elapsed;
    uint32_t speed
};
```

The **_media_playback** structure has been deprecated and replaced by **_media_playback_status**.

_media_playback_status

```
struct _media_playback_status {
    uint32_t  flags;
    uint32_t  state;
    uint32_t  speed;
    uint32_t  trkidx_total;
    uint32_t  trkidx_current;
    uint32_t  trkpos_total;
    uint32_t  trkpos_current;
    uint32_t  chpidx_total;
    uint32_t  chpidx_current;
    uint32_t  chppos_total;
    uint32_t  chppos_start;
    uint32_t  metaseq;
    uint32_t  reserved[4];
};
```

The **_media_playback_status** structure contains information about the current playback state of the device. It is returned when a DCMD_MEDIA_PLAYBACK message is sent to the control file. Any change to any element in this structure must trigger a notification event on the MediaFS control file. The **_media_playback_status** structure includes at least the members described in the table below.

For more information about possible values for playback states and flags values, see "Media playback constants" below.

Member	Type	Description
flags	**uint32_t**	Flags to indicate the playback speed status as well as other information about a media file. See "The *flags* and *speed* members" and "Media playback constants" below.
state	**uint32_t**	The current playback state of the device. Must be one of PLAYBACK_STATE_STOP, PLAYBACK_STATE_PLAY or PLAYBACK_STATE_PAUSE. This value must be updated on a device playback state change. See "Media playback constants" below.
speed	**uint32_t**	The playback speed. This value is valid only if the PLAYBACK_FLAG_FASTFWD or the PLAYBACK_FLAG_FASTRWD flag is set. See "The *flags* and *speed* members" below.
trkidx_total	**uint32_t**	The total number of tracks in the playback list.
trkidx_current	**uint32_t**	The index reference for the currently playing track.

continued...

Member	Type	Description
trkpos_total	**uint32_t**	The length of the currently playing track, in milliseconds. Set to 0 if the track length is not known.
trkpos_current	**uint32_t**	The current position in the currently playing track, in milliseconds.
chpidx_total	**uint32_t**	The total number of chapters in the current media item. Set to 0 (zero) if there are no chapters.
chpidx_current	**uint32_t**	The index reference for the currently playing chapter.
chppos_total	**uint32_t**	The length of the currently playing chapter, in milliseconds. Set to 0 (zero) if the chapter length is not known.
chppos_start	**uint32_t**	The offset, in milliseconds, from the start of the chapter from which to start playback of the chapter. Set to 0 (zero) if this offset is not known.
metaseq	**uint32_t**	A sequence number that changes if metadata values have changed during playback of the current track.
reserved [4]	**uint32_t**	Reserved for future use.

The *flags* and *speed* members

The value of the *flags* member can be a combination of:

- 0 (zero)
- PLAYBACK_FLAG_FASTFWD (0x01)
- PLAYBACK_FLAG_FASTRWD (0x02)
- PLAYBACK_FLAG_SPEED_EXACT (0x04)
- PLAYBACK_FLAG_EVENTS (0x08)
- PLAYBACK_FLAG_ALBART (0x10)
- PLAYBACK_FLAG_IS_VIDEO (0x20)

The PLAYBACK_FLAG_FASTFWD and PLAYBACK_FLAG_FASTRWD flag values are exclusive. If you set one, you must not set the other.

If *flags* is non-zero and the media device supports an indication of the exact playback speed, then PLAYBACK_FLAG_SPEED_EXACT flag can be set.

The *speed* member is updated on a playback speed change: 0 means paused, and 1 (one) means normal playback speed. The value of *speed* is only valid if the PLAYBACK_FLAG_SPEED_EXACT flag is set. If the

PLAYBACK_FLAG_SPEED_EXACT flag is not set in the *flags* member, *speed* should be set to 0 (zero).
You should combine the PLAYBACK_FLAG_* values to set the *flags* member.

See also “Media playback constants” below.

_media_settings

```
struct _media_settings {
    uint8_t value
};
```

The **_media_setting** structure is used in conjunction with the DCMD_MEDIA_GET_SHUFFLE, DCMD_MEDIA_SET_SHUFFLE, DCMD_MEDIA_GET_REPEAT and DCMD_MEDIA_SET_REPEAT device control messages. It contains the repeat or random mode setting for the device, and includes at least the members described in the table below.

Member	Type	Description
value	**uint8_t**	The repeat or random mode value for the device.

Separate messages must be issued for getting and setting random and repeat modes; that is, it is *not* possible to get or set both the random and the repeat mode with one *devctl()* call. See also “Repeat and random mode setting constants” below.

_media_speed

```
struct _media_speed {
    unsigned rate;
};
```

The **_media_speed** structure is used to set the current playback speed of the media device. The *rate* is a multiplication factor, where 1 (one) is normal playback speed. Valid values are 1, 2, 4, 8, 16 and 32.

This structure is used in conjunction with the DCMD_MEDIA_FASTFWD and DCMD_MEDIA_FASTRWD commands. It includes at least the members described in the table below.

Member	Type	Description
rate	unsigned	**The playback speed multiplication factor; 1 (one) is normal speed.**

_media_stream_info

```
struct _media_stream_info {
    unsigned char is_DRM;
    unsigned char seek_supported;
    unsigned char unused[2];
```

```
    uint32_t      reserved;
    uint64_t      stream_length;
};
```

The **_media_stream_info** structure is used to carry information that affects how a media stream can be played. It includes at least the members described in the table below:

Member	Type	Description
is_DRM	**char**	Indicate if the media stream is DRM (Digital Rights Management) protected. Set to either **Y** (protected) or **N** (not protected).
seek_supported	**char**	Indicate if the media stream supports seek capabilities. Set to either **Y** (supported) or **N** (not supported).
unused [2]	**char**	Reserved for future use.
reserved	**uint32_t**	Reserved for future use.
stream_length	**uint64_t**	The length of the media stream, in bytes. Set to MEDIA_STREAM_LENGTH_UNKNOWN if the media stream length is not known.

Playback constants

The tables below list the constants defined in **dcmd_media.h** for playback monitoring and control.

Media playback constants

The PLAYBACK_FLAG_* and PLAYBACK_STATE_* constants are defined in the structure **_media_playback_status**; they set or describe playback states.

Constant	Value	Description
PLAYBACK_FLAG_FASTFWD	0x01	Playback is in fast forward mode; the DCMD_MEDIA_FASTFWD control message has been applied, and playback speed is set to a number other than 1 (one).

continued...

Constant	Value	Description
PLAYBACK_FLAG_FASTRWD	0x02	Playback is in fast rewind mode; the DCMD_MEDIA_FASTRWD control message has been applied, and playback speed is set to a number other than 1 (one).
PLAYBACK_FLAG_SPEED_EXACT	0x04	The playback speed is the exact device speed; otherwise the playback speed is the value set with a DCMD_MEDIA_FAST*WD control message.
PLAYBACK_FLAG_EVENTS	0x08	Events are waiting to be retrieved from the event queue.
PLAYBACK_FLAG_ALBART	0x10	Album art is available to be read by a call with the DCMD_MEDIA_ALBART_READ control message.
PLAYBACK_FLAG_IS_VIDEO	0x20	Video is currently playing.
PLAYBACK_STATE_STOP	0	Playback is stopped.
PLAYBACK_STATE_PLAY	1	Playback is underway (not paused or stopped).
PLAYBACK_STATE_PAUSE	2	Playback is paused.

Repeat and random mode setting constants

The REPEAT_* and SHUFFLE_* constants set or describe playback repeat and random mode settings. The REPEAT_* values should be used with the DCMD_MEDIA_*_REPEAT messages, and the SHUFFLE_* should be used with the DCMD_MEDIA_*_SHUFFLE messages.

Constant	Value	Description
REPEAT_OFF	0	Repeat mode is off.
REPEAT_ONE_TRACK	1	Repeat the current track only.
REPEAT_ALL_TRACKS	2	Repeat all tracks.
REPEAT_FOLDER	3	Repeat all tracks in the folder.
REPEAT_SUBFOLDER	4	Repeat all tracks in the subfolder.
SHUFFLE_OFF	0	Random mode is off.

continued...

Constant	Value	Description
SHUFFLE_TRACKS	1	Play all tracks in pseudo-random order.
SHUFFLE_ALBUMS	2	Play all albums in pseudo-random order. The playback order of the tracks depends on whether SHUFFLE_TRACKS is set.
SHUFFLE_FOLDER	3	Play all tracks in the folder in pseudo-random order.
SHUFFLE_SUBFOLDER	4	Play all tracks in the subfolder in pseudo-random order.

Media stream constants

The MEDIA_STREAM_* constants set or describe media streams.

Constant	Value	Description
MEDIA_STREAM_LENGTH_UNKNOWN	-1	The media stream length is not known.

Media type strings

The table below lists common media type strings used in the **`info.xml`** file's **`<media>`**/**`<type>`** element to describe the mediastore. These mediastore types are consistent with the mediastore types defined by the MME's MME_STORAGETYPE_* constants in order to map type to string.

Constant	Value	Description
MEDIA_TYPE_UNKNOWN	"UNKNOWN"	Unknown storage type
MEDIA_TYPE_AUDIOCD	"AUDIOCD"	Audio CD
MEDIA_TYPE_VCD	"VCD"	Video CD
MEDIA_TYPE_SVCD	"SVCD"	Super Video CD
MEDIA_TYPE_FS	"FS"	RAM disc
MEDIA_TYPE_DVDAUDIO	"DVDAUDIO"	Audio DVD
MEDIA_TYPE_DVDVIDEO	"DVDVIDEO"	Video DVD
MEDIA_TYPE_IPOD	"IPOD"	iPod device
MEDIA_TYPE_KODAKCD	"KODAKCD"	Kodak picture CD

continued...

Constant	Value	Description
MEDIA_TYPE_PICTURECD	“PICTURECD”	Other picture CD
MEDIA_TYPE_A2DP	“A2DP”	A2DP protocol for Bluetooth
MEDIA_TYPE_SMB	“SMB”	MEDIA_TYPE_FS
MEDIA_TYPE_FTP	“FTP”	Internet FTP connection
MEDIA_TYPE_HTTP	“HTTP”	Internet HTTP connection
MEDIA_TYPE_NAVIGATION	“NAVIGATION”	Navigation CD or DVD.
MEDIA_TYPE_PLAYSFORSURE	“PFS”	PlaysForSure and similar devices.
MEDIA_TYPE_UPNP	“UPNP”	Devices using UPnP protocol.

iPod structures

MediaFS uses the following data structures to manage iPod devices:

- **`_media_ipod_daudio`**

`_media_ipod_daudio`

```
struct _media_ipod_daudio {
    unsigned  rate;
    int       sndchk;
    int       voladj;
    unsigned  reserved;
};
```

The **`_media_ipod_daudio`** structure is used to carry information about an iPod’s capabilities, and instructions to be applied to the iPod. It includes at least the following members:

Member	Type	Description
rate	**`unsigned`**	The sample rate, in Hertz, for the media on the device. Standard values are 32000, 44100 and 48000; some devices also support 8000, 11025, 12000, 16000, 22050 or 24000 Hertz.
sndchk	**`int`**	The device sound check value, as gain in decibels plus or minus. If the sound check capabilitiy is disabled on the device, this value must be set to 0.
voladj	**`int`**	The device volume adjustment, a gain in decibels plus or minus.

continued...

Member	Type	Description
reserved	`unsigned`	Reserved for future use.

Chapter 10

Getting Album Art

In this chapter. . .

MediaFS supports retrieval of album art associated with media files, if this capability is supported by the media device:

- How to retrieve album art
- Album art messages
- Album art structures
- Album art constants

How to retrieve album art

To retrieve album art associated with a media file, a high-level multimedia application, such as the MME, and the device driver must perform the following steps in sequence:

1 **Client application**: Issue a DCMD_MEDIA_ALBART_INFO message to the MediaFS **`control`** file, or to another specified file to find out if there is artwork associated with the file.

Device controller: Retrieve the required information from the device and return it in the **`_media_albart_entry`** data structure. If artwork is available, set the appropriate values in this structure's *flag* and *pos* members.

2 **Client application**: If artwork is available, issue a DCMD_MEDIA_ALBART_LOAD message.

Device controller: Complete and return the **`_media_albart_entry`** structure with the image description, so that the client application can know the size of the image and prepare to read it.

3 **Client application**: Issue DCMD_MEDIA_ALBART_READ messages to read the artwork and place it in a buffer, managing the returned image blocks and using them to reconstruct the image after the complete image has been read.

Device controller: Retrieve as requested the artwork in blocks from the media device, returning to the client application, as appropriate, one of:

- the number of bytes sent, if part of the image data was sent
- EAGAIN, if the device is still in the process of sending the image block and the client application needs to try again to get the next image block
- ENODATA, if the entire image has been read and there is no more data to send

Album art messages

To support album art retrieval, a device controller must support the following control messages from a higher-level application:

- DCMD_MEDIA_ALBART_INFO
- DCMD_MEDIA_ALBART_LOAD

- DCMD_MEDIA_ALBART_READ

If the queried media device does no support the album art retrieval, the *devctl()* call with the DCMD_MEDIA_ALBART_* message returns ENOTSUP.

DCMD_MEDIA_ALBART_INFO

The DCMD_MEDIA_ALBART_INFO message is used to query a media file for the presence of album artwork. The album art information for the file is placed in the **`_media_albart_entry`** data structure.

On success, a call to *devctl()* with this message returns EOK; the *devctl() dev_info_ptr* parameter points to the number or entries in the array with the album artwork.

Buffer: **`_media_albart_entry`**

DCMD_MEDIA_ALBART_LOAD

The DCMD_MEDIA_ALBART_LOAD message is used to retrieve the index information for a file whose album artwork is to be retrieved. The requested information is placed in the **`_media_albart_entry`** data structure.

On success, a call to *devctl()* with this message returns EOK; the *devctl() dev_info_ptr* parameter points to the index for the specified file.

Buffer: **`_media_albart_entry`**

DCMD_MEDIA_ALBART_READ

The DCMD_MEDIA_ALBART_READ message is used to read an album artwork image. The read process starts with the *devctl()* call with the DCMD_MEDIA_ALBART_READ message and ends when the entire image has been read.

Image data is read only once and returned; once a portion of an image has been read and returned, it is not returned again. The device controller must manage reading data blocks from the device, and the calling application must manage the returned data blocks until the entire image has read and can be passed up to an HMI application for display.

The total size of the image, in bytes, and other image information is placed in the **`_media_img_desc`** data structure.

When a call to *devctl()* with the DCMD_MEDIA_ALBART_READ message completes the device controller must return one of:

- ENODATA — the entire image has been read
- EAGAIN — the image is still being received

In addition, on completion of a successful call the *dev_info_ptr* parameter must point to the number of bytes received.

Buffer: **`_media_albart`**

Album art structures

MediaFS uses the following data structures to process the album art for media files:

- **_media_albart**
- **_media_albart_entry**
- **_media_img_desc**

_media_albart

```
struct _media_albart {
    uint32_t                flags;
    uint32_t                pos;
    uint32_t                reserved[6];
    struct _media_img_desc  desc;
    uint8_t                 data[1];
};
```

The **_media_albart** structure contains the album art data retrieved from a media file. It is populated and returned by *devctl()* when it successfully issues a DCMD_MEDIA_ALBART_READ message to a MediaFS file.

The data in this structure may not be the complete requested image, and multiple reads may be required to read a complete image. See DCMD_MEDIA_ALBART_READ in the chapter MediaFS Messages.

Member	Type	Description
flags	**uint32_t**	Flags specifying how to interpret position information for the album art. See the descriptions of the ALBART_FLAG_POS_* constants under “Album art constants” below.
pos	**uint32_t**	Position at which to display the album art. This position is either the offset, in milliseconds, in the track if ALBART_FLAG_POS_TRKPOS is set; or the chapter, if ALBART_FLAG_POS_CHPIDX is set.
reserved [6]	**uint32_t**	Reserved for future use.
desc	**struct**	The **_media_img_desc** structure with the image description.
data [1]	**uint8_t**	An array for the album art data.

_media_albart_entry

```
struct _media_albart_entry {
    uint16_t                index;
    uint16_t                reserved[3];
    uint32_t                flags;
    uint32_t                pos;
```

```
    struct _media_img_desc  desc;
};
```

Member	Type	Description
index	**uint16_t**	The index to match for this album art entry.
reserved [3]	**uint16_t**	Reserved for future use.
flags	**uint32_t**	Flags specifying how to interpret position information for the album art. See the descriptions of the ALBART_FLAG_POS_* constants under "Album art constants" below.
pos	**uint32_t**	Position at which to display the album art. This position is either the offset, in milliseconds, in the track if ALBART_FLAG_POS_TRKPOS is set; or the chapter, if ALBART_FLAG_POS_CHPIDX is set.
desc	**struct**	The **_media_img_desc** structure with the image description.

_media_img_desc

```
struct _media_img_desc {
    uint32_t  width;
    uint32_t  height;
    uint32_t  size;
    uint32_t  reserved;
    char      mimetype[64];
};
```

Member	Type	Description
width	**uint32_t**	The album art image width, in pixels.
height	**uint32_t**	The album art image height, in pixels.
size	**uint32_t**	The album art image size, in bytes.
reserved	**uint32_t**	Reserved for future use.
mimetype [64]	**char**	A string with the album art MIME type.

Album art constants

The table below lists the constants defined in **dcmd_media.h** for album artwork processing.

Constant	Value	Description
ALBART_FLAG_POS_NONE	0x00000000	No position information is available.
ALBART_FLAG_POS_TRKPOS	0x00000001	The position is expressed in milliseconds from the start of the track.
ALBART_FLAG_POS_CHPIDX	0x00000002	The position is the chapter number.
ALBART_FLAG_POS_MASK	0x0000000F	A mask for stripping out bits not relevant to the *flags* member of the **`_media_img_desc`** data structure.
ALBART_INDEX_NONE	0xFFFF	Indicate that no specific index is used, so that a call to *devctl()* with the DCMD_MEDIA_ALBART_LOAD message attempts to load the best match rather than a specific file.

Chapter 11

MediaFS Events

In this chapter. . .

This chapter describes MediaFS events and the data structures they use.

- Working with MediaFS events
- MediaFS event types
- MediaFS events and their structures

Working with MediaFS events

MediaFS supports events for communication between devices and upper-level applications. A device driver should, therefore, be designed to write, whenever the underlying device changes state, the appropriate MediaFS events and their payloads to the MediaFS event queue so that they can be read by client applications.

This section presents:

- The MediaFS event queue
- Reading MediaFS events

For a complete list of supported MediaFS event types, events and event data structures, see "MediaFS events and their structures" below.

The MediaFS event queue

The MediaFS event queue is the means by which a device driver can communicate playback status changes and updates, and device state changes to client applications in the sequence in which they occur.

The MediaFS event queue:

- is a fixed-size, circular queue
- implements FIFO (first in, first out) behavior

The MediaFS queue's FIFO behavior means that a client reading items from the queue will always receive events in chronological order.

Writing events to the queue

When the device controller writes an event to the MediaFS queue, it must:

- Set to PLAYBACK_FLAG_EVENTS the *flags* member of the **`_media_playback_status`** structure.
- Send an asynchronous notification to all clients registered on the **`control`** file.

Event queue management

The device controller should ensure the following event queue behavior:

- If the event queue is full when the device driver writes an event to it, the new event should overwrite the oldest event in the queue.

- When all items in the queue have been removed, the device controller should clear the PLAYBACK_FLAG_EVENTS flag in the **_media_playback_status** data structure's *flags* member.

In order to assure backwards compatibility with MediaFS 1.0, which did not support events, the MediaFS event queue is optional.

Reading MediaFS events

Multimedia applications using MediaFS should be designed to use *devctl()* calls with the DCMD_MEDIA_READ_EVENTS to read events from the MediaFS event queue, and to use the information provided by these events to manage media playback and other activities. To read MediaFS events, an application must call the *devctl()* function with the DCMD_MEDIA_READ_EVENTS message.

DCMD_MEDIA_READ_EVENTS

DCMD_MEDIA_READ_EVENTS instructs MediaFS to populate the client application's data buffer with data from the MediaFS event queue.

Buffer: **char[1]**

Managing your buffer when using DCMD_MEDIA_READ_EVENTS

The DCMD_MEDIA_READ_EVENTS is used to instruct MediaFS to populate the client application's data buffer with data from the MediaFS event queue. *It is the responsibility of the client application to ensure that it has a buffer large enough for the events in the MediaFS event queue*.

Behavior when the queue is larger than the client application buffer

If the number of bytes of data in the event queue is *greater than* the size of the client application's data buffer, a call to *devctl()* with the DCMD_MEDIA_READ_EVENTS message will:

- *not* write any data to the client application's data buffer
- set the **_media_event** structure's *len* member to the number of bytes *required* in the data buffer
- return EOK

In this case, the client application should:

1. Increase the size of the buffer it uses for the MediaFS events to at least the size returned in *len*.
2. Call *devctl()* with the DCMD_MEDIA_READ_EVENTS message again to read the events from the queue.

Behavior when the queue is smaller than or equal to the client application buffer

If the number of bytes of data in the event queue is *less than or equal* to the size of the client application's data buffer, a call to *devctl()* with the DCMD_MEDIA_READ_EVENTS message will:

- fill the buffer
- set the **_media_event** structure's *len* member to the number of bytes of data in the buffer to the number of bytes of data in the data buffer

Event types

MediaFS uses five types of events. Values for these event types are carried in the **_media_event** structure's *type* member. They are described in the table below:

Event type	Value	Description
MEDIA_EVENT_ERROR	0	Error
MEDIA_EVENT_WARNING	1	Warning
MEDIA_EVENT_TRACK	2	Communicate a track information change.
MEDIA_EVENT_TIME	3	Communicate a time update.
MEDIA_EVENT_METADATA	4	Communicate changes to metadata.

MediaFS events and their structures

This section describes MediaFS events, organized by event type. It includes:

- The **_media_event** data structure
- Track, time and other information update events
- Metadata update events
- Error and warning events

The _media_event data structure

```
_media_event
struct _media_event {
    uint32_t  type;
    uint32_t  len;
};
```

The **_media_event** structure is a included in all other MediaFS event structures. It specifies the event type, and the length of the event data. This structure includes at least the members described in the table below.

Member	Type	Description
type	**uint32_t**	The event type; see "Event types" above.
len	**uint32_t**	The length of the event data, in bytes (including padding to 8-byte alignment).

Track, time and other information update events

MediaFS information events signal an update to track or time information for the specified media track or file. Depending on the type of information they communicate, these events carry either the **_media_event_info**, the **_media_event_time** or the **_media_event_track** data structure.

The table below describes the MediaFS track and time update events:

Event	Value	Description
MEDIA_EVENT_INFO_UNKNOWN	0	Events carrying information, such as time or track updates, about a track or media file.

_media_event_info

```
struct _media_event_info {
    struct _media_event event;
    uint32_t            index;
    uint32_t            type;
    char                value[1];
};
```

The **_media_event_info** structure contains track or media file information. It should be populated when track or media file information changes, and, if relevant, included with the information events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
index	**uint32_t**	The index number for the track to which the event is associated.
type	**uint32_t**	The type of information event; see "Track, time and other information update events" above.
value [1]	**char**	A character string with the changed track information.

_media_event_time

```
struct _media_event_time {
    struct _media_event event;
    uint32_t            index;
    uint32_t            elapsed;
    uint32_t            duration;
};
```

The **_media_event_time** structure contains track or media file time information. It should be populated when track or media file time information changes, and, if relevant, included with the information events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
index	**uint32_t**	The index number for the track to which the event is associated.
elapsed	**uint32_t**	The elapsed time for the current track, in milliseconds.
duration	**uint32_t**	The track duration (total time) of the current track, in milliseconds.

_media_event_track

```
struct _media_event_track {
    struct _media_event event;
    uint32_t            index;
    uint32_t            duration;
    char                trackpath[1];
};
```

The **_media_event_track** structure contains track or media file information. It should be populated when track or media file information changes, and, if relevant, included with the information events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
duration	**uint32_t**	The track duration (total time) of the current track, in milliseconds.
trackpath	**char**	A character string with the path (relative to the mountpoint) of the current media file or track.

Metadata update events

MediaFS metadata events signal an update or other change to metadata for the specified media track or file. These events carry the data structure **`_media_event_metadata`**.
The table below describes the MediaFS metadata update events:

Event	Value	Description
MEDIA_EVENT_METADATA_UNKNOWN	0	An unspecified change has been made to the file's metadata.
MEDIA_EVENT_METADATA_SONG	1	Change to the file's song metadata.
MEDIA_EVENT_METADATA_ALBUM	2	Change to the file's album metadata.
MEDIA_EVENT_METADATA_ARTIST	3	Change to the file's artist metadata.
MEDIA_EVENT_METADATA_GENRE	4	Change to the file's genre metadata.
MEDIA_EVENT_METADATA_COMPOSER	5	Change to the file's composer metadata.
MEDIA_EVENT_METADATA_RELEASE_DATE	6	Change to the file's release date metadata.
MEDIA_EVENT_METADATA_TRACK_NUM	7	Change to the file's track number metadata.
MEDIA_EVENT_METADATA_PUBLISHER	8	Change to the file's publisher metadata.
MEDIA_EVENT_METADATA_DURATION	9	Change to the file's duration metadata.
MEDIA_EVENT_METADATA_NAME	10	Change to the file's name metadata.
MEDIA_EVENT_METADATA_COMMENT	11	Change to the file's comment metadata.

_media_event_metadata

```
struct _media_event_metadata {
    struct _media_event event;
    uint32_t            type;
    uint32_t            index;
    uint32_t            duration;
    struct _media_date  date;
    char                value[1];
};
```

The **_media_event_metadata** structure contains track metadata. It should be populated whenever metadata for a track or media file changes, and included with the metadata update events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
type	**uint32_t**	The type of metadata event; see "Metadata update events" above.
index	**uint32_t**	The index number for the track to which the event is associated.
duration	**uint32_t**	The track date.
date	**struct**	The **_media_date** structure with the track date information.
value [1]	**char**	A UTF-8 encoded character string for character-based metadata types.

Error and warning events

MediaFS error and warning events signal an error or other condition that requires attention from the client application. These events carry, respectively, the data structure **_media_event_error** or **_media_event_warning**.

The table below describes the MediaFS error and warning events:

Event	Value	Description
MEDIA_EVENT_ERROR_UNKNOWN	0	An unspecified error condition has occurred.
MEDIA_EVENT_ERROR_DRM	1	A DRM error has occurred.
MEDIA_EVENT_ERROR_CORRUPT	2	The media file is corrupt.

continued...

Event	Value	Description
MEDIA_EVENT_WARNING_UNKNOWN	0	An unspecified condition that requires attention has occurred.

_media_event_error

```
struct _media_event_error {
    struct _media_event  event;
    uint32_t             index;
    uint32_t             type;
};
```

The **_media_event_error** structure contains track or media file error information. It should be populated when an error is encountered with a track or media file, and included with the error events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
index	**uint32_t**	The index number for the track to which the event is associated.
type	**uint32_t**	The type of error event; see “Error and warning events” above.

_media_event_warning

```
struct _media_event_warning {
    struct _media_event  event;
    uint32_t             index;
    uint32_t             type;
};
```

The **_media_event_warning** structure contains track or media file error information. It should be populated when a warning situation is encountered with a track or media file, and included with the error events that MediaFS places in its event queue.

Member	Type	Description
event	**struct**	The **_media_event** structure with the event type and size.
index	**uint32_t**	The index number for the track to which the event is associated.

continued...

Member	Type	Description
type	`uint32_t`	The type of warning event; see "Error and warning events" above.

Appendix A

MediaFS Examples

In this appendix. . .

This appendix presents some examples that help illustrate how to use MediaFS. It contains:

- MediaFS structure
- **info.xml** file

MediaFS structure

The following presents a MediaFS instance representing an iPod device:

```
ipod0/:
total 3
dr-xr-xr-x  3 root      root      512 Jun 01 11:28 .
dr-xr-xr-x  2 root      root        0 Jun 01 11:28 ..
dr-xr-xr-t  3 root      root      512 Jun 01 11:28 .FS_info.
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Music

ipod0/.FS_info.:
total 6
dr-xr-xr-t  3 root      root      512 Jun 01 11:28 .
dr-xr-xr-x  3 root      root      512 Jun 01 11:28 ..
nrw-rw-rw-  1 root      root        0 Jun 01 11:28 control
lrwxrwxrwx  1 root      root        0 Jun 01 11:28 current ->
-r--r--r--  1 root      root     1127 Jun 01 11:28 info.xml
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 playback

ipod0/.FS_info./playback:
total 2
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 .
dr-xr-xr-t  3 root      root      512 Jun 01 11:28 ..

ipod0/Music:
total 10
dr-xr-xr-x 10 root      root      512 Jun 01 11:28 .
dr-xr-xr-x  3 root      root      512 Jun 01 11:28 ..
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Albums
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Artists
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Audiobooks
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Composers
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Genres
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Playlists
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Podcasts
dr-xr-xr-x  2 root      root      512 Jun 01 11:28 Songs
```

info.xml file

The following presents an MediaFS **info.xml** file for an iPod device:

```
<?xml version="1.0" standalone="yes"?>
<info>
```

```
<media>
    <device>iPod</device>
    <protocol>
        <general>1.02</general>
        <display_remote>1.01</display_remote>
        <extended>1.09</extended>
    </protocol>
    <name>Yov Yovchev&#x2019;s iPod</name>
    <serial>JQ44915UR5S</serial>
    <swversion>1.2.1</swversion>
    <model>
        <id>0x00060000</id>
        <number>P9585LL</number>
        <generation>1</generation>
        <type>iPod photo</type>
        <size>40GB</size>
        <color>white</color>
    </model>
    <audio>
        <eq>off</eq>
    </audio>
    <display>
        <limit>
            <type>2</type>
            <format>le_rgb565</format>
            <height>110</height>
            <width>210</width>
        </limit>
        <limit>
            <type>3</type>
            <format>be_rgb565</format>
            <height>110</height>
            <width>210</width>
        </limit>
        <limit>
            <type>1</type>
            <format>mono</format>
            <height>110</height>
            <width>210</width>
        </limit>
    </display>
</media>
<fsys>
    <type>ipod</type>
    <mountpoint>/fs/ipod0</mountpoint>
    <mountdevice>file-2-ipod-5-media</mountdevice>
</fsys>
<device>
    <driver>ipod</driver>
    <catagory>media</catagory>
    <transport>
        <type>ser_ipod</type>
```

```
        <dev>/dev/serfpga3</dev>
    </transport>
</device>
</info>
```

For an example from an **info.xml** file used for a media changer device, see "The **info.xml** file for mediastore changers" in the chapter Media Changers.

Index

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

W