QNX® Aviage Multimedia Suite 1.2.0

MME Technotes

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 May 13, 2009.

Contents

About this Reference

The *MME Technotes* accompanies the QNX Aviage multimedia suite, release 1.2.0. It is intended for application developers who require additional, specific reference materials for their multimedia projects.
The table below may help you find what you need in this book:

For information about:	See:
The QNX PFS driver access function provided with the QNX Aviage Multimedia Interface for PlaysForSure.	User-specified MTP commands to PFS devices
How to set up the MME ADE (Audio Decoder Engine) to work with the Texas Instruments ADE 5.1.3.	MME support for Texas Instruments ADE

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 Utilities	Utilities used by the MME.
MME Configuration Guide	How to configure the MME.
MediaFS Developer's Guide	Developer's guide for implementing MediaFS.
QDB Developer's Guide	QDB database engine programming guide and API library reference.

Note that the MME is a component of the QNX Aviage multimedia core package, which is available in the QNX Aviage multimedia suite of products. The MME is the main component of this core package. It is used for configuration and control of your multimedia applications.

Typographical conventions

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

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	**`make`**
Environment variables	**PATH**
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	Ctrl-Alt-Delete
Keyboard input	**`something you type`**
Keyboard keys	Enter
Program output	**`login:`**
Programming constants	NULL
Programming data types	**`unsigned short`**
Programming literals	`0xFF`, **`"message string"`**
Variable names	*stdin*
User-interface components	**Cancel**

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

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

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

Notes point out something important or useful.

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

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

Note to Windows users

In our documentation, we use a forward slash (`/`) as a delimiter in *all* pathnames, including those pointing to Windows files.

We also generally follow POSIX/UNIX filesystem conventions.

Technical support options

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

Chapter 1

User-specified MTP Commands to PFS Devices

In this chapter. . .

This document describes *PFS_ExecuteCommand()*, the QNX PFS driver access function provided with the QNX Aviage Multimedia Interface for PlaysForSure. This function provides client application developers:

- a single function to handle non-POSIX-based, user-specified MTP (Media Transfer Protocol) commands to a PFS device
- extensibility, by allowing user applications to support new MTP commands without having to update their system's PFS driver

PFS_ExecuteCommand()

Support user-specified MTP commands to a PFS device

Synopsis

```
#include <pfs_userx.h>

MTP_RESULT PFS_ExecuteCommand( int fd,
                               OPCODE opCode,
                               int nSendParams,
                               MTP_UINT32 *pSendParams,
                               int handleParamNumber,
                               int nSendDataBytes,
                               MTP_UINT8 *pSendDataBytes,
                               int *pnRecvParams,
                               MTP_UINT32 *pRecvParams,
                               int *pnRecvDataBytes,
                               MTP_UINT8 *pRecvDataBytes,
                               RESPONSECODE *pResponseCode );
```

Arguments

fd — An open file descriptor that determines the PFS device to which the command is sent. The opened file must be a file in the PFS file system.

opCode — The 16-bit MTP operation code (see MTP specification).

nSendParams — The number of 32-bit operation parameters to send with the designated operation code. The maximum value is 5.

pSendParams — The address of an array of 32-bit operation parameters. This address is the source of the operation parameters. The number of operation parameters is indicated by *nSendParams*. If *nSendParams* is 0, then *pSendParams* may be NULL.

handleParamNumber

Use this parameter to have one of the operation parameters automatically set to the MTP object handle associated with the open file. Set to 0 to disable this feature. Set to 1 to cause the

first operation parameter to be overwritten by the MTP handle (in general, *pSendParms*[*handleParamNumber*-1] = *handle*).

nSendDataBytes The number of data bytes to send in the transfer phase. If there is no data to send, set to 0.

pSendDataBytes The address of the buffer with the data bytes to send in the data transfer phase. If there is no data to send, this argument must be set to NULL.

pnRecvParams A pointer to an integer specifying the maximum number of parameters expected. During input, this number is the maximum number of receive parameters expected. During output, this argument is updated to the actual number of parameters received. The maximum value for this parameter is 5.

pRecvParams The address of an array to hold the received 32-bit response parameters. The the number of bytes written in this array is four times either the maximum number of parameters expected or the actual number of parameters received, whichever is less.

pnRecvDataBytes A pointer to an integer specifying the maximum number of data bytes expected in the data transfer phase of the operation. During input, this number is the maximum number of data bytes to receive. During output, this number is updated to the actual number of data bytes transfer to the receive buffer. If there is no data to receive, this argument must be set to NULL.

pRecvDataBytes The address of the buffer to receive data transferred from the PFS device. If there is no data to receive, this argument must be set to NULL.

pResponseCode A pointer to the location where the response code received from the device is written.

Library

pfs_userx.h

Description

The function *PFS_ExecuteCommand()* allows applications to execute atomic MTP on PFS devices. An atomic MTP command is executed as a single transaction, which consists of the following:

1. Send the operation code and parameters to the device.
2. Optionally, transfer data to or from the device.
3. Receive a response code and response parameters from the device.

PFS_ExecuteCommand() is not intended for reading of media content (i.e. execute OPCODE_GETOBJECT), and its data transfers are expected to be less than 64 kilobytes. To read media content, use the POSIX *read()* function.

- The source for *PFS_ExecuteCommand()* is available with the function, in the file **`pfs_readme.txt`**.
- *PFS_ExecuteCommand()* is *not* included in any QNX object libraries.

Events

None delivered.

Blocking and validation

This function validates that at least one of the data buffers (read or write) is set to NULL. It executes to completion.

Returns

MTP_RESULT_OK	Success.
MTP_ERROR_*	An error occurred. See MTP error codes below for more information.
< 0	An error occurred in a call to the system function *MsgSendv()*, indicating a communication error with the PFS driver (return is -*errno*).

MTP error codes

PFS_ExecuteCommand() relays to the client application any errors it receives from the PFS driver. The function does some parameter validation, and only explicitly returns the MTP_ERROR_INVALIDARG error code; all other error codes are generated by either the PFS driver or the PFS device itself.

The enumerated values listed below define the most common errors returned or relayed by *PFS_ExecuteCommand()*. For a complete list, see the MTP specification.

- MTP_RESULT_SPECIFIC — The operation was *not* successful. Check the response code for more information about what type of error occurred.
- MTP_ERROR_BAD_TRANSACTION_ID — the response transaction ID was incorrect.
- MTP_ERROR_DATATYPE_MISMATCH — the expected transfer phase did not occur, or the transfer phase occurred when it was not expected.
- MTP_ERROR_DEVICE_NOT_CONNECTED — the PFS device has been disconnected.

- MTP_ERROR_EMPTY — the expected data transfer did not occur.
- MTP_ERROR_INVALIDARG — the two buffers specified (send and receive), or some other parameter value is not valid.
- MTP_ERROR_IO_INCOMPLETE — a USB transaction did not complete for an unspecified reason.
- MTP_ERROR_MTP_SPECIFIC — An MTP error occurred. Check the value referenced by *pResponseCode* for more information.
- MTP_ERROR_OUTOFMEMORY — the PFS driver was unable to allocate the memory needed for the receive buffer.
- MTP_ERROR_READ_FAULT — a USB read stalled (see the USB specification).
- MTP_ERROR_TIMEOUT — a USB operation took too long.
- MTP_ERROR_WRITE_FAULT — a USB write stalled (see the USB specification).

Classification

QNX Neutrino

Safety	
Interrupt handler	No
Signal handler	No
Thread	Yes

Examples

The following examples show how you can use *PFS_ExecuteCommand()* to execute:

- an MTP command with no data transfer
- an MTP command with a transfer from the device
- an MTP command with a transfer to the device

```
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <sys/iomsg.h>
#include <fcntl.h>
#include "pfs_userx.h"

#ifndef MTP_RESULT_OK
#define MTP_RESULT_OK 0
#endif

int main(int argc, char *argv[])
{
```

```
char         *music = "/fs/pfs0/Music";
int           i, n, fd;
MTP_RESULT    hr;
MTP_UINT32    send_params[5];
MTP_UINT32    recv_params[5];
MTP_UINT32    nRecvParams = 1;
MTP_UINT32    nRecvDataBytes = 0;
MTP_UINT8    *pSendDataBytes;
MTP_UINT8    *pRecvDataBytes;
RESPONSECODE ResponseCode = 0;
```

MTP command with no data transfer

```
// example 1: no data transfer, get number of objects under Music folder
fd = open(music, O_RDONLY);
if (fd > 0) {

    send_params[0] = 0xffffffff;  // StorageID
    send_params[1] = 0;           // ObjectFormat
    send_params[2] = 0;           // ObjectHandle (pararameter 3 to be overwritten with handle)
    nRecvParams = 1;

    hr = PFS_ExecuteCommand(fd, 0x1006,    // OPCODE_GETNUMOBJECTS

                        3,            // number of 32- bit send parameters
                        send_params,  // address of array of 32-bit send parameters

                        3,            // operation parameter to contain the object handle

                        0,            // number of data bytes to send in transfer phase
                        NULL,         // address of buffer with data bytes to send

                        &nRecvParams, // input: maximum number of receive parameters expected,
                                      // output: actual received
                        recv_params,  // address of array to hold response parameters

                        NULL,     // input: max data bytes to receive, output: actual received
                        NULL,     // address of buffer to receive data

                        &ResponseCode);     // the response code received

    printf("\nexample 1: hr=%d, RC=%04x, num_recv_parms=%d, num_objects=%d\n",
            hr, ResponseCode, nRecvParams, recv_params[0]);
    close(fd);
}
```

MTP command with a transfer from the device

```
// example 2: receive data, get object handles from Music directory
fd = open(music, O_RDONLY);
pRecvDataBytes = (MTP_UINT8 *) alloca(n = sizeof(MTP_UINT32) * 1000);
if (pRecvDataBytes != NULL && fd > 0) {

    send_params[0] = 0xffffffff;  // StorageID
    send_params[1] = 0;           // ObjectFormat
    send_params[2] = 0;           // ObjectHandle (pararameter 3 to be overwritten with handle)
    nRecvParams = 0;
    nRecvDataBytes = n;

    hr = PFS_ExecuteCommand(fd, 0x1007,   // OPCODE_GETOBJECTHANDLES
```



```
                        3,                  // number of 32-bit send parameters
                        send_params,        // address of array of 32-bit send parameters

                        3,                  // operation parameter to contain the object handle

                        0,                  // number of data bytes to send in transfer phase
                        NULL,               // address of buffer with data bytes to send

                        NULL,               // input: maximum number of receive parameters expected,
                                            // output: actual received
                        NULL,               // address of array to hold response parameters

                        &nRecvDataBytes,    // input: maximum data bytes to receive, output:
                                            // actual received
                        pRecvDataBytes,     // address of buffer to receive data

                        &ResponseCode);     // the response code received

    printf("\nexample 2: hr=%d, RC=%04x, num_recv_bytes=%d\n", hr, ResponseCode, nRecvDataBytes);
    n = (nRecvDataBytes - sizeof(MTP_UINT32)) / sizeof(MTP_UINT32);
    if (hr == MTP_RESULT_OK && n != 0 && n == pRecvDataBytes[0]) {
        for (i = 1; i <= n; ++i)
            printf("\n  %3d %#10x", i, ((MTP_UINT32 *) pRecvDataBytes)[i]);
    }
    printf("\n");
    close(fd);
}
```

MTP command with a transfer to the device

```
// example 3: send data, update the "Friendly Name" device property
fd = open(music, O_RDONLY);
n = 5;  // write five UNICODE16 characters
pSendDataBytes = (MTP_UINT8 *) alloca(2*(n+1));
if (pSendDataBytes != NULL && fd > 0) {

    send_params[0] =  0xD402;       // DEVICEPROPCODE_DEVICEFRIENDLYNAME
    pSendDataBytes[0] =  n;         // "NAME" with null has total of 5 characters
    pSendDataBytes[1] = 'N';        // 16-bit characters
    pSendDataBytes[2] =  0;         // - high bits are zero
    pSendDataBytes[3] = 'A';
    pSendDataBytes[4] =  0;
    pSendDataBytes[5] = 'M';
    pSendDataBytes[6] =  0;
    pSendDataBytes[7] = 'E';
    pSendDataBytes[8] =  0;
    pSendDataBytes[9] =  0;         // 16-bit null
    pSendDataBytes[10] = 0;

    hr = PFS_ExecuteCommand(fd, 0x1016,   // OPCODE_SETDEVICEPROPVALUE

                        1,                  // number of 32-bit send parameters
                        send_params,        // address of array of 32-bit send parameters

                        0,                  // operation parameter to contain the object handle

                        1+2*n,              // number of data bytes to send in transfer phase
                        pSendDataBytes,     // address of buffer with data bytes to send

                        NULL,               // input: maximum number of receive parameters expected,
```

```
                                        // output: actual received
                            NULL,       // address of array to hold response parameters

                            NULL,       // input: maximum data bytes to receive, output: actual received
                            NULL,       // address of buffer to receive data

                            &ResponseCode);   // the response code received

        printf("\nexample 3: hr=%d, RC=%04x\n", hr, ResponseCode);
        close(fd);
    }
    return 0;
}
```

Chapter 2

MME Support for Texas Instruments ADE

In this chapter. . .

This document describes how to set up the MME ADE (Audio Decoder Engine) to work with the Texas Instruments ADE. The Texas Instruments ADE supports decoding of files in WMA9, AAC, MP3, PCM and WAV formats. It runs on the Texas Instruments Jacinto EVM.

Overview

When your system uses the ADE, a writer in the MME sends the compressed format to the ADE running on the DSP for decoding. The decoded PCM is then sent directly from the DSP to one of the three DACs on the Jacinto EVM.

For more information about the MME, see the *Introduction to the MME*, and the *MME Developer's Guide*. For more information about the Texas Instruments ADE, see the documentation provided by the manufacturer.

Requirements

The Texas Instruments ADE release package contains the following required items:

- **`Audio_init`** — a binary that initializes the external audio codecs using I2C and SPI.
- **`loaddspimage`** — a binary that loads and starts the DSP image.
- **`audio_app.out`** — a DSP executable.

The **`dsplink`** 1.40.05 package contains:

- **`CFG_Jacinto.c`** — specific configuration file for building **`dsplink`** 1.40.05 for the ADE writer.

The **`dsplink`** 1.61 package contains:

- **`CFG_ARM.c`** and **`CFG_DRA44XGEM_SHMEM.c`** — specific configuration files for building **`dsplink`** 1.61 for the ADE writer.

You must also ensure that your installation of **`io-media-generic`** includes **`ade3_writer.so`**. This writer is only available for ARMLE targets.

Version compatibility

The tables below shows component compatibility, and where components can be obtained. Component versions listed in a table are compatible *only* with other components listed in the same table:

- MME 1.1 with **`dsplink`** 1.40.05 patch 3.0
- MME 1.1 with **`dsplink`** 1.40.05 patch 3.3
- MME 1.1 with **`dsplink`** 1.61
- MME 1.2 with **`dsplink`** 1.61

MME 1.1 with `dsplink` 1.40.05 patch 3.0

The table below lists the components required for the MME 1.1 with **`dsplink`** 1.40.05 patch 3.0.

Component	Version	Availability
`dsplink`	1.40.05 patch 3.0	Foundry27 — BSPs
`ade3_writer.so`	Compiled for **`dsplink`** 1.40.05 patch 3.0.	Foundry27 — included in MME 1.1
TI ADE	1.01.00 — compiled for **`dsplink`** 1.40.05 patch 3.0.	Request from Texas Instruments.

MME 1.1 with `dsplink` 1.40.05 patch 3.3

The table below lists the components required for the MME 1.1 with **`dsplink`** 1.40.05 patch 3.3.

Component	Version	Availability
`dsplink`	1.40.05 patch 3.3	Foundry27 — BSPs
`ade3_writer.so`	Compiled for **`dsplink`** 1.40.05 patch 3.3.	Request from QNX.
TI ADE	1.01.03 — compiled for **`dsplink`** 1.40.05 patch 3.3.	Request from Texas Instruments.

MME 1.1 with `dsplink` 1.61

The table below lists the components required for the MME 1.1 with **`dsplink`** 1.61.

Component	Version	Availability
`dsplink`	1.61	Foundry27 — BSPs
`ade3_writer.so`	Compiled for **`dsplink`** 1.61.	Request from QNX.
TI ADE	1.02.02 — compiled for **`dsplink`** 1.61.	Request from Texas Instruments.

MME 1.2 with `dsplink` 1.61

The table below lists the components required for the MME 1.2 with **`dsplink`** 1.61.

Component	Version	Availability
`dsplink`	1.61	Foundry27 — BSPs

continued...

Component	Version	Availability
ade3_writer.so	Compiled for **dsplink** 1.61.	Foundry27 — included in MME 1.2
TI ADE	1.02.02 — compiled for **dsplink** 1.61.	Request from Texas Instruments.

The QNX Foundry27 web site is at: **www.foundry27.com**.

Installation

Installation of the ADE requires the following tasks:

- Build the **dsplink** resource manager
- Build the Jacinto image
- Modify the MME for Jacinto

For installation instructions, see the **dsplink** project Installation Notes on Foundry27.

Build the dsplink resource manager

To build the **dsplink** resource manager:

1. For **dsplink** 1.40.05 *only*, replace the QNX **CFG_Jacinto.c** found in **/lib/dsplink14005/config/all/** with the Texas Instruments **CFG_Jacinto.c** from the **64M** directory.
   For **dsplink** 1.6*n only*, replace the **CFG_ARM.c** and **CFG_DRA44XGEM_SHMEM.c** files in **lib/dsplink160/config/all/** with the configuration files from the Texas Instruments release package.
2. For all **dsplink** versions, build the following to create the **dsplink** resource manager:
   - **/lib/dsplink14005**, **/lib/dsplink161**or other **dsplink** version, as required
   - **/services/dsplink**

   This build changes the ARM/DSP memory split.

Build the Jacinto image

To build the Jacinto image:

1. Ensure that the build has the 64-megabyte memory configuration for **dsplink**, by commenting out the default configuration line:

   ```
   #Startup for dsplink config 8M ---CURRENT DEFAULT---
   #startup-jacinto -L 0x67800000,0x800000 -vvvvv
   ```

   and uncommenting the line for the 64-megabyte memory configuration:

   ```
   #Startup for dsplink config 64M
   startup-jacinto  -L 0x64000000,0x4000000 -vvvv
   ```

2. Initialize the drivers — most of this information is also available in the documentation supplied by Texas Instruments with the ADE release:

```
i2c-dm6446 -p0x01C21000 -i39
waitfor /dev/i2c0
i2c-dm6446 -p0x01C21800 -i41 --u 2
waitfor /dev/i2c2
spi-master -d dra446 base=0x01c24c00,irq=37,edmairq=0xc128,edmachannel=40,edma=1
waitfor /dev/spi0
regaccess -v0xE08cc5F1 -p0x01C48100 -l32
regaccess -v0x114cc450 -p0x01C48104 -l32
regaccess -v0x022aa02f -p0x01C48108 -l32
regaccess -v0xffffffff -p0x01c4800c -l32
regaccess -v0x0F000000 -p0x01C48034 -l32
regaccess -v0x00000018 -p0x01C4812C -l32
regaccess -v0x12       -p0x18000020 -l32
regaccess -v0x12       -p0x18001020 -l32
Audio_init -v
```

3. Start **dsplink**:

```
/proc/boot/dsplink &
waitfor /dev/dsplink
```

4. Load the DSP image:

```
loaddspimage /proc/boot/audio_app.out &
```

5. Make sure the following binaries are available to the system:

- **regaccess**
- **dsplink**
- **loaddspimage**
- **audio_app.out**
- **Audio_init**

Modify the MME for Jacinto

To modify the MME for Jacinto support, do the following:

1. Modify the **mme_data.sql** file to *not* use the default configuration by commenting out the following lines, as shown below:

```
-- This example configures one output device in one zone, for one control context
--INSERT INTO outputdevices(type, permanent, name, devicepath)
--    VALUES(1, 1, 'defaultoutput', '/dev/snd/pcmC0D1p');
--INSERT INTO zones(zoneid, name) SELECT 1, 'defaultzone';
--INSERT INTO zoneoutputs(zoneid, outputdeviceid)
--    SELECT 1, outputdeviceid FROM outputdevices
--    WHERE name='defaultoutput';
--INSERT INTO renderers(path) VALUES('/dev/io-media');
--INSERT INTO controlcontexts(zoneid, rendid, name)
--    VALUES( 1, 1, 'default' );
```

2. Modify the **mme_data.sql** file to use the Jacinto multi-zone configuration example by uncommenting or adding, as required, the following:

```
-- Jacinto multi-zone configuration example.
-- Three Control Contexts, three Zones, one output device each.
-- One io-media for all:
INSERT INTO renderers(path)
    VALUES('/dev/io-media');

-- zone0:
INSERT INTO outputdevices(type, permanent, name, devicepath)
    VALUES(1, 1, 'appe_0', 'appe_output_0');
INSERT INTO zones(zoneid, name)
    SELECT 1, 'zone0';
INSERT INTO zoneoutputs(zoneid, outputdeviceid)
    SELECT 1, outputdeviceid FROM outputdevices
        WHERE name='appe_0';
INSERT INTO controlcontexts(zoneid, rendid, name)
    VALUES( 1, 1, 'cc0' );

-- zone1:
INSERT INTO outputdevices(type, permanent, name, devicepath)
    VALUES(1, 1, 'appe_1', 'appe_output_1');
INSERT INTO zones(zoneid, name)
    SELECT 2, 'zone1';
INSERT INTO zoneoutputs(zoneid, outputdeviceid)
    SELECT 2, outputdeviceid FROM outputdevices
        WHERE name='appe_1';
INSERT INTO controlcontexts(zoneid, rendid, name)
    VALUES( 2, 1, 'cc1' );

-- zone2:
INSERT INTO outputdevices(type, permanent, name, devicepath)
    VALUES(1, 1, 'appe_2', 'appe_output_2');
INSERT INTO zones(zoneid, name)
    SELECT 3, 'zone2';
INSERT INTO zoneoutputs(zoneid, outputdeviceid)
    SELECT 3, outputdeviceid FROM outputdevices
        WHERE name='appe_2';
INSERT INTO controlcontexts(zoneid, rendid, name)
    VALUES( 3, 1, 'cc2' );
```

3 In the QDB configuration file (**qdb.cfg**), use **tmpfs** for **mme_library**, **mme_temp** and **mme**. For example:

```
[mme_temp]
Filename    = /fs/tmpfs/mme_temp.db
Schema File = /db/mme_temp.sql
```

Startup

To start the MME, use the standard MME startup procedure described in the MME Quickstart Guide in *Introduction to the MME*, but start **io-fs-media** as follows:

```
# io-fs-media -d tmp,noglob -cpages=4 -cbundles=0
```

and start **io-media-generic** as follows:

```
io-media-generic -Mmmf,dlldir=$QNX_TARGET/armle/lib/dll/mmedia \
-Mmmf,audio_writer=ade3_writer -Mmmf,keepdlls=all
```

When using **mmecli**, you need to specify which control context to use: cc0, cc1 or cc2. Each control context plays to a different headphone output port on the Jacinto EVM. For example, to play to two different output zones do the following:

```
mmecli -c /dev/mme/cc0 newtrksession l "select fid from library"
mmecli -c /dev/mme/cc0 settrksession 1
mmecli -c /dev/mme/cc0 play 1
mmecli -c /dev/mme/cc1 newtrksession l "select fid from library"
mmecli -c /dev/mme/cc1 settrksession 2
mmecli -c /dev/mme/cc1 play 2
```

Note that the **newtrksession** option is the letter “l”, for a library mode track session; while the **settrksession** option is the numeral one, which is the track session number.

Audio routing

Audio routing is outside the scope of the MME for controlling the APPE audio routing, mixing, and post processing on the DSP. You must develop a separate audio management application to support these capabilities.