palmsource™

Multimedia

Exploring Palm OS®

Written by Christopher Bey
Edited by Jean Ostrem
Technical assistance from Eric Moon and Marco Nelissen

Exploring Palm OS: Multimedia
Document Number 3112-002
November 9, 2004
For the latest version of this document, visit
http://www.palmos.com/dev/support/docs/.

PalmSource, Inc.
1240 Crossman Avenue
Sunnyvale, CA 94089
USA
www.palmsource.com

Table of Contents

About This Document

This book describes the portions of Palm OS® that provide multimedia capabilities. This includes the Sound Manager and the Multimedia Library.

IMPORTANT: The *Exploring Palm OS* series is intended for developers creating native applications for Palm OS Cobalt. If you are interested in developing applications that work through PACE and that also run on earlier Palm OS releases, read the latest versions of the *Palm OS Programmer's API Reference* and *Palm OS Programmer's Companion* instead.

Who Should Read This Book

You should read this book if you are a Palm OS software developer and you want to do one of the following:

- Play simple, monophonic sounds such as beeps or alerts in an application.
- Play or record stereo, sampled sounds in an application.
- Write an application that plays or records audio-visual media.

Beginning Palm OS developers may want to delay reading this book until they gain a better understanding of the fundamentals of Palm OS application development. Instead, consider reading *Exploring Palm OS: Programming Basics* to gain a good understanding of event management and *Exploring Palm OS: User Interface* to learn about events generated by standard UI controls. Come back to this book when you find you need to use the sound and multimedia services.

What This Book Contains

This book contains the following information:

- Part I, “Sound Manager,” contains information on the Sound Manager:
  - Chapter 1, “Sound Manager,” on page 3, describes how to use the Sound Manager to play and record sound.
  - Chapter 2, “Sound Manager Reference,” on page 11, describes the Sound Manager API.
- Part II, “Multimedia Library,” contains information on the Multimedia Library:
  - Chapter 3, “Multimedia Applications,” on page 53, describes how to use the Multimedia Library to play and record multimedia content.
  - Chapter 4, “Multimedia Library Definitions,” on page 69, describes common Multimedia Library API elements.
  - Chapter 5, “Multimedia Codecs,” on page 79, describes the Multimedia Library API related to codecs.
  - Chapter 6, “Multimedia Formats,” on page 83, describes the Multimedia Library API related to formats.
  - Chapter 7, “Multimedia Properties,” on page 109, describes the Multimedia Library API related to properties.
  - Chapter 8, “Multimedia Sessions,” on page 115, describes the Multimedia Library API related to sessions.
  - Chapter 9, “Multimedia Tracks,” on page 149, describes the Multimedia Library API related to tracks.
- “Glossary” on page 155, is a glossary of multimedia terms.

Changes to This Book

3112-002

- Minor bug fixes.

3112-001

- Initial version.

The *Exploring Palm OS* Series

This book is a part of the *Exploring Palm OS* series. Together, the books in this series document and explain how to use the APIs exposed to third-party developers by the fully ARM-native versions of Palm OS, beginning with Palm OS Cobalt. Each of the books in the *Exploring Palm OS* series explains one aspect of the Palm operating system and contains both conceptual and reference documentation for the pertinent technology.

As of this writing, the complete *Exploring Palm OS* series consists of the following titles:

- *Exploring Palm OS: Programming Basics*
- *Exploring Palm OS: Memory, Databases, and Files*
- *Exploring Palm OS: User Interface*
- *Exploring Palm OS: User Interface Guidelines* (coming soon)
- *Exploring Palm OS: System Management*
- *Exploring Palm OS: Text and Localization*
- *Exploring Palm OS: Input Services*
- *Exploring Palm OS: High-Level Communications*
- *Exploring Palm OS: Low-Level Communications*
- *Exploring Palm OS: Telephony and SMS*
- *Exploring Palm OS: Multimedia*
- *Exploring Palm OS: Security and Cryptography*
- *Exploring Palm OS: Creating a FEP* (coming soon)
- *Exploring Palm OS: Application Porting Guide*

Additional Resources

- Documentation

  PalmSource publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  PalmSource and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/support/kb/

Part I
Sound Manager

The Sound Manager plays and records sound.

1

Sound Manager

The Palm OS® Sound Manager controls two independent sound facilities:

- **Simple sound**: Single voice, monophonic, square-wave sound synthesis, useful for system beeps and the like. This works on all version of Palm OS.
- **Sampled sound**: Stereo, multi-format, sampled data recording and playback. Sampled sounds can be generated programmatically or read from a sound file.

These facilities are independent of each other. Although you can play a simple sound and a sampled sound at the same time, their respective APIs have no effect on each other. For example, you can't use the sampled sound volume-setting function (`SndStreamSetVolume()`) to change the volume of a simple sound.

The following sections take a look at the concepts introduced by the Sound Manager. For detailed API descriptions, and for more guidance with regard to the sampled data concepts presented here, see Chapter 2, "Sound Manager Reference."

Note that the Multimedia Library documented in Part II, "Multimedia Library," can also play sounds. Compared to the Sound Manager, the Multimedia Library is more complex to use, but provides more robust features and supports more sound formats, such as MP3, which is not supported by the current Sound Manager implementation. The Sound Manager is probably better to use if you want to play short or frequent sounds with a lightweight facility.

Simple Sound

There are three ways to play a simple sound:

- You can play a single tone of a given pitch, amplitude, and duration by calling `SndDoCmd()`.
- You can play a pre-defined system sound ("Information," "Warning," "Error," and so on) with `SndPlaySystemSound()`.
- You can play a tune by passing in a Level 0 Standard MIDI File (SMF) through the `SndPlaySmf()` function. For example, the alarm sounds used in the built-in Date Book application are MIDI records stored in the System MIDI database. For information on MIDI and the SMF format, go to the official MIDI website, `http://www.midi.org/` .

Sampled Sound

In the sampled sound facilities, there are two fundamental functions:

- `SndStreamCreate()` opens and configures a new sampled sound "stream" from/into which you record/playback buffers of "raw" data. An alternate function, `SndStreamCreateExtended()`, lets you use variable-sized buffers.
- `SndPlayResource()` plays sound data that's read from a (formatted) sound file. The function configures the playback stream for you, based on the format information in the sound file header. Currently, only uncompressed WAV and IMA ADPCM WAV formats are recognized. (Note that IMA ADPCM is also known as DVI ADPCM). `SndPlayResource()` is *only* used to play back sound; it can't be used for recording.

The Sound Manager also provides functions that let you set the volume (`SndStreamSetVolume()`) and stereo panning (`SndStreamSetPan()`) for individual recording and playback streams.

Simple vs. Sampled Sound

Comparing the two facilities, simple sound is easy to understand and requires very little programming: In most cases, you load up a structure, call a function, and out pops a beep. Unfortunately, the sound itself is primitive. (An example of simple sound programming is given in “Sound Preferences,” below.)

Sampled sound, on the other hand, is (or can be) much more satisfying, but requires more planning than simple sound. How much more depends on what you’re doing. Playing samples from a sound file isn’t much more difficult than playing a simple sound, but you have to supply a sound file. Generating samples programmatically—and recording sound—requires more work: You have to implement a callback function that knows something about sound data.

IMPORTANT: One significant difference between simple sounds and sampled sounds is that they use different volume scales: Simple sound volumes are in the range [0, 64]; sampled sound volumes are in the range [0, 1024].

Sound Preferences

If you’re adding short, “informative” sounds to your application, such as system beeps, alarms, and the like, you should first consider using the (simple) system sounds that are defined by Palm OS in the `SndSysBeepType` enum and played by `SndPlaySystemSound()`.

If you want to create your own system-like sounds, you should at least respect the user’s preferences settings with regard to sound volume. There are three sound preference constants:

- `prefSysSoundVolume` is the default system volume.
- `prefGameSoundVolume` is used for game sounds.
- `prefAlarmSoundVolume` is used for alarms.

To apply a sound preference setting to a simple sound volume, you have to retrieve the setting and apply it yourself. For example, in Listing 1.1 we retrieve the alarm sound and use it to set the volume of a simple sound.

Listing 1.1 Playing a simple sound with the alarm volume

```
// Create a 'sound command' structure. This will encode the parameters of the
// tone we want to generate.
SndCommandType sndCommand;

// Ask for the 'play a tone' command.
sndCommand.cmd = sndCmdFreqDurationAmp;

// Set the frequency and duration.
sndCommand.param1 = 1760;
sndCommand.param2 = 500;

// Now get the alarm volume and set it in the struct.
sndCommand.param3 = PrefGetPreference (prefAlarmSoundVolume);

// Play the tone.
SndDoCmd( 0, &sndCommand, true);
```

The sampled sound API, on the other hand, provides volume constants (`sndSystemVolume`, `sndGameVolume`, and `sndSysVolume`) that look up a preference setting for you, as shown in Listing 1.2.

Listing 1.2 Playing a sampled sound with the alarm volume

```
// Point our sound data pointer to a record that contains WAV data (record
// retrieval isn't shown).
SndPtr soundData = MemHandleLock(...);

// Play the data using the default alarm volume setting.
SndPlayResource(soundData, sndAlarmVolume, sndFlagNormal);

// Unlock the data.
MemPtrUnlock(soundData);
```

Standard MIDI Files

Although you can use a Level 0 Standard MIDI File to control simple sound generation, this doesn't imply broad support for MIDI messages; only key down, key up, and tempo change messages are recognized.

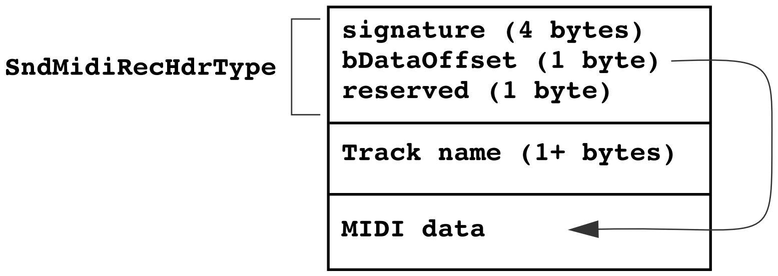
You can store your MIDI data in a MIDI database:

- The database type `sysFileTMidi` identifies MIDI record databases.
- The system MIDI database is further identified by the creator `sysFileCSystem`. The database holds a number of system alarm sounds.

You can add MIDI records to the system MIDI database, or you can store them in your own.

Each record in a MIDI database is a concatenation of a PalmSource-defined MIDI record header, the human-readable name of the MIDI data, and then the MIDI data itself. Figure 1.1 depicts a complete Palm OS MIDI record.

Figure 1.1 Palm OS Midi Record



To get to the track name, use an expression like this:

```
pName = (char*)hdrP + sndMidiRecHdrSize;
```

The MIDI track name is `null`-terminated, even if it's empty. It's at least one byte long and at most `sndMidiNameLength` bytes long.

The code in Listing 1.3 creates a new MIDI record and adds it to the system MIDI database.

Listing 1.3 Adding a new MIDI record

```
// We need three things: A header, a name, and some data. We'll get the name
// and data from somewhere, and create the header ourselves.
char *midiName = ...;
MemHandle midiData = ...;
SndMidiRecHdrType midiHeader;
```

```
// Database and record gadgetry.
DmOpenRef database;
MemHandle record;
uint16_t *recordIndex = dmMaxRecordIndex;
uint8_t* recordPtr;
uint8_t* midiPtr;

// MIDI header values: Always set the signature to sndMidiRecSignature, and
// reserved to 0. bDataOffset is an offset from the beginning of the header to
// the first byte of actual MIDI data. The name includes a null-terminator,
// hence the '+ 1'.
midiHeader.signature = sndMidiRecSignature;
midiHeader.reserved = 0;
midiHeader.bDataOffset = sizeof(SndMidiRecHdrType) + StrLen(midiName) + 1;

// Open the database and allocate a record.
database = DmOpenDatabaseByTypeCreator( sysFileTMidi, sysFileCSystem,
               dmModeReadWrite | dmModeExclusive);
record = DmNewRecord( database, &recordIndex,
               midiHeader.bDataOffset + MemHandleSize(midiData));

// Lock the data and the record.
midiDataPtr = MemHandleLock(midiData);
recordPtr = MemHandleLock(record);

// Write the MIDI header.
DmWrite( recordPtr, 0, &smidiHeader, sizeof(midiHeader));

// Write the track name.
DmStrCopy( recordPtr, sndMidiRecHrdSize, midiName);

// Write the MIDI data.
DmWrite( recordPtr, midiHeader.bDataOffset, midiDataPtr, MemHandleSize(midiData));

// Unlock the handles, release the record, close the database.
MemHandleUnlock( midiData);
MemHandleUnlock( record);
DmReleaseRecord( database, recordIndex, 1);
DmCloseDatabase( database);
```

To retrieve a MIDI record, you can use the `SndCreateMidiList()` function if you know the record's creator, or you can use the Data Manager functions to iterate through all MIDI records.

Creating a Sound Stream

The sound stream API, part of the sampled sound facility, is the most flexible part of the Sound Manager. A sound stream sends sampled data to or reads sampled data from the sound hardware. There are several sound output streams and one input stream, all running (or potentially running) concurrently.

NOTE: The maximum number of output streams is dependent on system resources. The default number is 18, but device manufacturers can change that.

To use a sound stream, you have to tell it what sort of data you're going to give it or that you expect to get from it. All of the sound format information that you need to supply to set up the stream—data quantization, sampling rate, channel count, and so on—is passed in the `SndStreamCreate()` or `SndStreamCreateExtended()` function.

You also have to pass the function a pointer to a callback function (see `SndStreamBufferCallback()` or `SndStreamVariableBufferCallback()`); implementing this function is where you'll be doing most of your work. When you tell your stream to start running (`SndStreamStart()`), the callback function is called automatically, once per buffer of data. If you're operating on an input stream (in other words, if you're recording), your callback function can do something with the data and then should return before the next buffer shows up. Output stream callbacks do the opposite—they fill the buffer with data.

Because of the real-time nature of audio playback, the callbacks must operate as quickly as possible. There can be more than one stream competing for attention. Note that all callbacks run in their own threads.

The formats that are supported by the sampled sound functions are described in the functions themselves.

Summary of Sound Manager

Simple Sound Functions	
SndCreateMidiList()	SndPlaySmfResource()
SndDoCmd()	SndPlaySystemSound()
SndGetDefaultVolume()	SndGetDefaultVolume()
SndPlaySmf()	SndSetDefaultVolume()

Sampled Sound Functions	
SndPlayResource()	SndStreamPause()
SndStreamCreate()	SndStreamSetPan()
SndStreamCreateExtended()	SndStreamSetVolume()
SndStreamDelete()	SndStreamStart()
SndStreamGetPan()	SndStreamStop()
SndStreamGetVolume()	

2

Sound Manager Reference

This chapter describes the Sound Manager API. It covers:

The header files `SoundMgr.h` and `AudioTypes.h` declare the API that this chapter describes.

For more information on the Sound Manager, see Chapter 1, "Sound Manager."

Sound Manager Structures and Types

SndCallbackInfoType Struct

Purpose Encapsulates a callback function and its argument data. `SndCallbackInfoType` is used by the `SndSmfCallbacksType` structure, which is used to list the callback functions that are called during SMF playback.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndCallbackInfoType {
   MemPtr funcP;
   uint32_t dwUserData;
} SndCallbackInfoType
```

Fields `funcP`

A pointer to the callback function.

`dwUserData`
: Data that's passed as an argument to the callback function.

SndCommandType Struct

Purpose Encapsulates a sound synthesis operation and its associated parameters. It is used by the `SndDoCmd()` function.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndCommandType {
   SndCmdIDType cmd;
   uint8_t reserved;
   uint16_t padding;
   int32_t param1;
   uint16_t param2;
   uint16_t param3;
} SndCommandType
typedef SndCommandType *SndCommandPtr
```

Fields

`cmd`
: Constant that represents a sound operation. The operations are listed and described in `SndDoCmd()`.

`reserved`
: Reserved for future use.

`padding`
: Padding bytes.

`param1, param2, param3`
: Operation-specific parameters. The parameters' meanings are described in `SndDoCmd()`.

SndMidiListItemType Struct

Purpose Locates a MIDI file. This structure is used by the `SndCreateMidiList()` function.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndMidiListItemType {
   char name[sndMidiNameLength];
   uint32_t uniqueRecID;
   DatabaseID dbH;
} SndMidiListItemType
```

Fields

`name`
: The null-terminated name of the MIDI file.

`uniqueRecID`
: The ID of the record that holds the MIDI file.

`dbH`
: Database ID of the database that holds the record.

SndMidiRecHdrType Struct

Purpose Encapsulates the header of a MIDI record.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndMidiRecHdrType {
   uint32_t signature;
   uint8_t bDataOffset;
   uint8_t reserved;
   uint16_t padding;
} SndMidiRecHdrType
```

Fields

`signature`
: The MIDI record signature. Always set this field to `sndMidiRecSignature`.

`bDataOffset`
: Offset, in bytes, from the beginning of the record to the first byte of the MIDI data.

`reserved`
: Reserved for future use. Always set this field to 0.

`padding`
: Padding bytes (not counted toward the record size).

SndPtr Typedef

Purpose Used to cast a pointer to the sound data used by `SndPlayResource()`.

Declared In `SoundMgr.h`

Prototype

```
typedef void *SndPtr
```

SndSampleType Typedef

Purpose Used to specify the sample format (size, data type, endianness) of a sampled sound stream. Used by `SndStreamCreate()`. See "audio_type_t" on page 17 for the set of values that this type can contain.

Declared In `SoundMgr.h`

Prototype

```
typedef audio_type_t SndSampleType
```

SndSmfCallbacksType Struct

Purpose Contains a set of application-defined functions that are called during MIDI playback. To register your callback functions, call `SndPlaySmf()`.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndSmfCallbacksType {
   SndCallbackInfoType completion;
   SndCallbackInfoType blocking;
   SndCallbackInfoType reserved;
} SndSmfCallbacksType
```

Fields

`completion`
Completion function; see `SndComplFuncType()`.

`blocking`
Blocking function; see `SndBlockingFuncType()`.

`reserved`
Reserved. Set this field to 0.

SndSmfChanRangeType Struct

Purpose Defines the range of enabled MIDI channels. Events on MIDI channels outside the enabled range are ignored. By default, no channels are enabled.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndSmfChanRangeType {
   uint8_t bFirstChan;
   uint8_t bLastChan;
} SndSmfChanRangeType
```

Fields

`bFirstChan`
The first enabled channel in the range [0, 15].

`bLastChan`
The last enabled channel in the range [0, 15].

IMPORTANT: The `SndSmfChanRangeType` structure expects MIDI channels to be in the range [0, 15]; real MIDI channel values are in the range [1, 16]. Thus, PalmSource MIDI channel 0 is real MIDI channel 1, PalmSource MIDI channel 1 is real MIDI channel 2, and so on.

SndSmfOptionsType Struct

Purpose Defines MIDI performance parameters.

Declared In `SoundMgr.h`

Prototype

```
typedef struct SndSmfOptionsType {
   uint32_t dwStartMilliSec;
   uint32_t dwEndMilliSec;
   uint16_t amplitude;
   Boolean interruptible;
   uint8_t reserved1;
   uint32_t reserved;
} SndSmfOptionsType
```

Fields

`dwStartMilliSec`
The “beginning of performance” marker, measured in milliseconds from the beginning of the track. A value of 0 plays the track from the beginning. The time difference

between `dwStartMilliSec` and the performance time of the first subsequent MIDI event is respected. For example, if `dwStartMilliSec` is 2000 and the first (subsequent) note-on event is at 3000, there will be a 1000 millisecond "pause" before the note is played.

`dwEndMilliSec`
: The "end of performance" marker, measured in milliseconds from the beginning of the track. To play to the end of the track, set this to `sndSmfPlayAllMilliSec`.

`amplitude`
: The volume of the track, in the range [0, `sndMaxAmp`]. The default is `sndMaxAmp`. If set to 0, the MIDI file isn't played.

`interruptible`
: If `true` (the default), MIDI playback is interrupted if the user interacts with the controls (digitizer, buttons, etc.), even if the interaction doesn't generate a sound command. If `false`, playback is not interrupted.

`reserved1`
: Reserved.

`reserved`
: Reserved. Set this field to 0.

Comments This structure is used with the `SndPlaySmf()` function to establish new parameter settings or to return the currently set values, depending on how the function is called. In the case where the structure returns values, only the "performance marker" fields (`dwStartMilliSec` and `dwEndMilliSec`) are valid.

SndStreamRef Typedef

Purpose Represents a sampled stream. You create an `SndStreamRef` with `SndStreamCreate()`.

Declared In `SoundMgr.h`

Prototype `typedef uint32_t SndStreamRef`

Sound Manager Constants

audio_type_t Enum

Purpose Defines a set of constants that represent the sample format (size, data type, endianness) of a sampled sound stream. These constant values are used with `SndStreamCreate()` and `SndStreamCreateExtended()`, as the value of the type `SndSampleType`.

The lower four bits of these constants gives the size (in bytes) of a single sample, as shown here:

```
uint8_t byteSize = formatConstant & 0x0f
```

Declared In `AudioTypes.h`

Constants

`sndInt8 = 0x01`
Signed 8-bit data.

`sndUInt8 = 0x11`
Unsigned 8-bit data.

`sndInt16Big = 0x02`
Signed 16-bit integer data in big-endian format.

`sndInt16Little = 0x12`
Signed 16-bit integer data in little-endian format.

`sndInt32Big = 0x04`
Signed 32-bit integer data in big-endian format.

`sndInt32Little = 0x14`
Signed 32-bit integer data in little-endian format.

`sndFloatBig = 0x24`
Signed floating-point data in big-endian format.

`sndFloatLittle = 0x34`
Signed floating-point data in little-endian format.

`sndInt16 = sndInt16Little`
Signed 16-bit integer data in the device's native endianness.

`sndInt16Opposite = sndInt16Big`
: Signed 16-bit integer data in the endianness opposite to that of the device.

`sndInt32 = sndInt32Little`
: Signed 32-bit integer data in the device's native endianness.

`sndInt32Opposite = sndInt32Big`
: Signed 32-bit integer data in the endianness opposite to that of the device.

`sndFloat = sndFloatLittle`
: Signed floating-point data in the device's native endianness.

`sndFloatOpposite = sndFloatBig`
: Signed floating-point data in the endianness opposite to that of the device.

Comments In the current implementation the 32-bit and floating point formats aren't supported.

Simple Sound Amplitudes

Purpose These constants can be supplied to the simple sound functions (such as `SndDoCmd()`) when an amplitude value is required. Note that these values are not compatible with the sampled sound amplitude range and thus shouldn't be used with the sampled sound functions.

Declared In `SoundMgr.h`

Constants `#define sndDefaultAmp sndMaxAmp`
: The maximum amplitude (full volume).

`#define sndMaxAmp 64`
: The default amplitude.

SndCmdIDType Enum

Purpose Contains constants that represent specific sound operations used in simple sound playback with `SndDoCmd()`.

Declared In `SoundMgr.h`

Constants `sndCmdFreqDurationAmp = 1`
Play a tone. `SndDoCmd()` blocks until the tone has finished.

`sndCmdNoteOn`
Initiate a MIDI-defined tone. `SndDoCmd()` returns immediately while the tone plays in the background. Subsequent sound playback requests interrupt the tone.

`sndCmdFrqOn`
Initiate a tone. `SndDoCmd()` returns immediately while the tone plays in the background. Subsequent sound playback requests interrupt the tone.

`sndCmdQuiet`
Stop the playback of the currently generated tone.

SndFormatType Enum

Purpose Defines a set of constants that represent various sound data encoding formats. Pass one of these constants as the *`format`* argument to `SndStreamCreateExtended()`.

Declared In `SoundMgr.h`

Constants `sndFormatPCM = 0`
Pulse Code Modulation format. This is the "no encoding" format; the data is a series of samples that are linear with regard to amplitude quantization and regular with regard to sampling rate.

`sndFormatIMA_ADPCM = 'APCM'`
The Interactive Multimedia Association's implementation of "adaptive delta" encoding. The sampling rate is constant, but the quantization is non-linear.

`sndFormatDVI_ADPCM = 'DPCM'`
Microsoft's adaptive delta implementation. This is the same as IMA ADPCM.

`sndFormatMP3 = 'MPG3'`
Motion Picture Group Audio Layer III.

`sndFormatAAC = 'DAAC'`
Dolby Advanced Audio Coding.

`sndFormatOGG = 'OGGV'`
OGG Vorbis encoding.

Comments The implementation of `SndStreamCreateExtended()` supports `sndFormatPCM` data only. To play ADPCM data, use `SndPlayResource()`.

sndMidiNameLength

Purpose Defines the maximum string length, including the null terminator, for the name of a MIDI file or MIDI track.

Declared In `SoundMgr.h`

Constants `#define sndMidiNameLength 32`

sndMidiRecHdrSize

Purpose Defines the header size of a MIDI record.

Declared In `SoundMgr.h`

Constants `#define sndMidiRecHdrSize 6`

sndMidiRecSignature

Purpose Tags a MIDI record. It is used as the value of the `signature` field of the `SndMidiRecHdrType` structure.

Declared In `SoundMgr.h`

Constants `#define sndMidiRecSignature 'PMrc'`

sndSmfPlayAllMilliSec

Purpose Represents the (temporal) far end of a MIDI file. You can use this constant as the value of the `dwEndMilliSec` field of the

SndSmfOptionsType structure before passing the structure to SndPlaySmf(). This setting tells the function to play the entire file.

Declared In `SoundMgr.h`

Constants `#define sndSmfPlayAllMilliSec 0xFFFFFFFFUL`

SndSmfCmdEnum Enum

Purpose Defines a set of commands that tell SndPlaySmf() whether it should play the file or simply return the duration of the file in milliseconds.

Declared In `SoundMgr.h`

Constants

`sndSmfCmdPlay = 1`
: Play the specified Standard MIDI File.

`sndSmfCmdDuration`
: Return the duration, in milliseconds, of the specified Standard MIDI File.

SndStreamMode Enum

Purpose Defines constants that represent the “direction” (input or output) of a sampled sound stream. Use these constants with the SndStreamCreate() function.

Declared In `SoundMgr.h`

Constants

`sndInput`
: Input stream used for recording.

`sndOutput`
: Output stream used for playback.

SndStreamWidth Enum

Purpose Defines constants that represent mono and stereo sampled data streams. Use these constants with the SndStreamCreate() function.

Declared In `SoundMgr.h`

Constants

`sndMono`
Mono (one channel) stream.

`sndStereo`
Stereo (two channel) stream.

SndSysBeepType Enum

Purpose Defines a set of constants that represent pre-defined system beeps. In order to play one of these sounds, pass the corresponding value to `SndPlaySystemSound()`.

Declared In `SoundMgr.h`

Constants

`sndInfo = 1`
Signals non-crucial information.

`sndWarning`
Grabs the user's attention.

`sndError`
Indicates an illegal operation.

`sndStartUp`
Played at device start up time.

`sndAlarm`
Generic alarm sound; note that this is *not* the Datebook's alarm sound.

`sndConfirmation`
Indicates approval or acceptance.

`sndClick`
The button click sound.

`sndCardInserted`
Played when a card is inserted.

`sndCardRemoved`
Played when a card is removed.

Sound Error Codes

Purpose Error codes returned by various Sound Manager functions.

Declared In `SoundMgr.h`

Constants

`#define sndErrBadChannel (sndErrorClass | 2)`
Invalid sound channel.

`#define sndErrBadParam (sndErrorClass | 1)`
Invalid parameter passed to a function.

`#define sndErrBadStream (sndErrorClass | 8)`
Invalid data stream.

`#define sndErrFormat (sndErrorClass | 7)`
Unsupported data format.

`#define sndErrInterrupted (sndErrorClass | 9)`
Play was interrupted.

`#define sndErrInvalidStream (sndErrorClass | 11)`
Invalid stream identifier.

`#define sndErrMemory (sndErrorClass | 3)`
Insufficient memory.

`#define sndErrNotImpl (sndErrorClass | 10)`
Function not implemented

`#define sndErrOpen (sndErrorClass | 4)`
Tried to open a channel that's already open.

`#define sndErrQEmpty (sndErrorClass | 6)`
Internal error.

`#define sndErrQFull (sndErrorClass | 5)`
The sound queue is full.

Sound Resource Playback Flags

Purpose Use these flags when calling `SndPlayResource()` to specify various settings. Currently, the only setting is function synchronization.

Declared In `SoundMgr.h`

Constants

`#define sndFlagSync 0x00000000`
: Tells `SndPlayResource()` to wait until all sound data has been fed to the DAC before returning (meaning that the function will return just a bit before the sound has finished playing).

`#define sndFlagAsync 0x00000001`
: Tells `SndPlayResource()` to return immediately while playback continues in a separate thread.

`#define sndFlagNormal sndFlagSync`
: A shorthand for the set of “normal” flag settings.

Sound Stream Feature Constants

Purpose Used to retrieve the Sound Manager version number from the Feature Manager.

Declared In `SoundMgr.h`

Constants

`#define sndFtrIDVersion 0`
: The feature number to supply to `FtrGet()`, along with a creator ID of `sysFileCSoundMgr`, when attempting to obtain the version of the Sound Manager.

`#define sndMgrVersionNum (100)`
: The current version of the Sound Manager. Note that in Palm OS Cobalt version 6.0, this is set incorrectly to 100 (indicating version 1.00). The Sound Manager is actually version 1.01 in this release.

Stereo Pan Constants

Purpose Define the extremes and the midpoint when altering the stream’s stereo balance with `SndStreamSetPan()`. `SndStreamSetPan()`

allows you to set the balance to one of these values, or any integral value between `sndPanFullLeft` and `sndPanFullRight`.

Declared In `SoundMgr.h`

Constants

`#define sndPanCenter (0)`
The stereo balance is centered.

`#define sndPanFullLeft (-1024)`
The stereo balance is panned completely to the left.

`#define sndPanFullRight (1024)`
The stereo balance is panned completely to the right.

Volume Constants Enum

Purpose Use the volume constants defined in this enum with `SndStreamSetVolume()` and `SndPlayResource()`. The constants tell the functions to retrieve the named sound volume preference (as set by the user) and apply it as a volume setting.

Declared In `SoundMgr.h`

Constants

`sndSystemVolume = -1`
The user's system sound preference.

`sndGameVolume = -2`
The user's game sound preference.

`sndAlarmVolume = -3`
The user's alarm sound preference.

Sound Manager Functions and Macros

SndCreateMidiList Function

Purpose Generates a list of MIDI records.

Declared In `SoundMgr.h`

Prototype
```
Boolean SndCreateMidiList (uint32_t creator,
   Boolean multipleDBs, uint16_t *wCountP,
   MemHandle *entHP)
```

Parameters

→ *`creator`*
Creator ID of the database in which the function looks for MIDI records. Pass 0 to search all databases.

→ *`multipleDBs`*
Pass `true` to search multiple databases for MIDI records. Pass `false` to search only in the first database that meets the search criteria.

← *`wCountP`*
Returns the number of MIDI records that were found.

← *`entHP`*
Returns a pointer to an array of `SndMidiListItemType` structures, one structure for each record that was found.

Returns `true` if records were found, `false` otherwise.

SndDoCmd Function

Purpose Asks the Sound Manager to perform a simple sound synthesis operation.

Declared In `SoundMgr.h`

Prototype
```
status_t SndDoCmd (void *channelP,
   SndCommandPtr cmdP, Boolean noWait)
```

Parameters

→ *`channelP`*
Pointer to the sound channel on which you want to perform the operation. Pass `NULL` for the “shared” sound channel.

IMPORTANT: The Sound Manager only supports one channel of sound synthesis: You must pass `NULL` as the value of *`channelP`*.

→ *`cmdP`*
Pointer to a `SndCommandType` structure that describes the operation and contains any associated parameters. See the Comments section below for the set of sound commands and their associated parameters.

→ *`noWait`*
Sets the function to be asynchronous (`true`) or synchronous (`false`) with respect to the caller.

IMPORTANT: `SndDoCmd()` is always synchronous: The *`noWait`* value is currently ignored.

Returns `errNone` if the operation completed successfully, or one of the following if an error occurs:

`sndErrBadParam`
Invalid parameter.

`sndErrBadChannel`
Invalid channel pointer.

`sndErrQFull`
The sound queue is full.

Comments The sound operations that are performed by `SndDoCmd()` are encapsulated in the `SndCommandType` structure. The `cmd` field represents the operation, while the `param` fields are data that's passed to the operation. The operations and data that `SndDoCmd()` supports are described in the following table.

Table 2.1 SndDoCmd() commands and parameters

Command	Function	Parameters
`sndCmdFreqDurationAmp`	Plays a tone. `SndDoCmd()` blocks until the tone has finished.	`param1` is the tone's frequency in Hertz. `param2` is its duration in milliseconds. `param3` is its amplitude in the range [0, `sndMaxAmp`]. If the amplitude is 0, the sound isn't played and the function returns immediately.
`sndCmdFrqOn`	Initiates a tone. `SndDoCmd()` returns immediately while the tone plays in the background. Subsequent sound playback requests interrupt the tone.	`param1` is the tone's frequency in Hertz. `param2` is its duration in milliseconds. `param3` is its amplitude in the range [0, `sndMaxAmp`]. If the amplitude is 0, the sound isn't played and the function returns immediately.
`sndCmdNoteOn`	Initiates a MIDI-defined tone. `SndDoCmd()` returns immediately while the tone plays in the background. Subsequent sound playback requests interrupt the tone.	`param1` is the tone's pitch given as a MIDI key number in the range [0, 127]. `param2` is the tone's duration in milliseconds. `param3` is its amplitude given as MIDI velocity [0, 127].
`sndCmdQuiet`	Stops the playback of the currently generated tone.	All parameter values are ignored.

See Also `SndPlaySmf()`

SndGetDefaultVolume Function

Purpose Returns volume levels cached by the Sound Manager. This function is deprecated and should not be used.

Declared In `SoundMgr.h`

Prototype
```
void SndGetDefaultVolume (uint16_t *alarmAmpP,
    uint16_t *sysAmpP, uint16_t *masterAmpP)
```

Parameters

← *alarmAmpP*
Pointer to the alarm amplitude.

← *sysAmpP*
Pointer to the system sound amplitude.

← *masterAmpP*
Pointer to the master amplitude.

Returns Nothing.

Comments Pass `NULL` for those settings that you don't care about.

Never call this function. To retrieve default volume levels, you should ask for the user's preferences settings.

See Also `SndSetDefaultVolume()`

SndPlayResource Function

Purpose Plays formatted sound data read from a resource or file.

Declared In `SoundMgr.h`

Prototype
```
status_t SndPlayResource (SndPtr sndP,
    int32_t volume, uint32_t flags)
```

Parameters

→ *sndP*
A pointer to the beginning of the formatted sound (including the header). Currently, only WAV data is recognized (see the Comments section, below); in this case, *sndP* must point to the "RIFF" ID (byte 0 in a simple `.wav` file).

→ *volume*
Amplitude scalar, in the range [0, 32k]. See `SndStreamSetVolume()` for information on how amplitude scalar values are applied.

→ *`flags`*
: One of the "Sound Resource Playback Flags" on page 24. Currently, the only setting is function synchronization: use `sndFlagSync` to have the function wait until all sound data has been fed to the DAC before returning, or `sndFlagAsync` flag to have the function return immediately while playback continues in a separate thread.

Returns `errNone` if the operation completed successfully, or one of the following if an error occurs:

`sndErrBadParam`
: The specified resource or file contains no data.

`sndErrFormat`
: The data is in an unsupported format.

`sndErrMemory`
: The function couldn't allocate sufficient memory.

other errors
: The device couldn't allocate system resources for the sound.

Comments The supported WAVE parameters are:

- Uncompressed (PCM) or IMA 4-bit adaptive differential (IMA ADPCM). The ADPCM type is also known as DVI ADPCM; in a WAV file, it's known as format 0x11.
- One or two-channels
- Any sampling rate

You can't interrupt or abort a resource playback once it's been initiated. The resource always plays to the end of the data.

Example The following code excerpt shows how to use this function to play a sound resource.

```
SndPtr soundP;
MemHandle recordH;
recordH = DmGetResource(myOpenDb,sysResTSound,TestWaveSound);
soundP = (SndPtr) MemHandleLock(recordH);
SndPlayResource(soundP,1024,sndFlagSync);
                // 1024 is 0dB (unity) gain
MemHandleUnlock(recordH);
```

The above code first gets the resource from an open database. It then locks the memory associated with the resource and converts the result to a pointer to a sound resource.

The call to the `SndPlayResource()` function plays the sound. The second parameter is the sound level, which varies from 0 to 32767. A value of 1024 specifies unity gain. Higher values indicate higher gain. The third parameter specifies whether the function returns immediately or waits until after the sound has finished playing. You should avoid using `sndFlagAsync`, which causes the function to return immediately without waiting for the sound to finish, if you use this code because you'll unlock the sound resource before the system finishes with it. In fact, you should always specify `sndFlagSync` unless you can guarantee two things:

1. Your resource memory remains locked for the duration of the sound.
2. Your application does not exit for the duration of the sound.

Remember that you cannot stop a sound played with `SndPlayResource()`.

SndPlaySmf Function

Purpose Performs a Standard MIDI File, or returns the duration of the file.

Declared In `SoundMgr.h`

Prototype

```
status_t SndPlaySmf (void *chanP,
    SndSmfCmdEnum cmd, uint8_t *smfP,
    SndSmfOptionsType *selP,
    SndSmfChanRangeType *chanRangeP,
    SndSmfCallbacksType *callbacksP,
    Boolean bNoWait)
```

Parameters

→ *`chanP`*
A pointer to the sound channel on which you want to perform the MIDI file. Pass `NULL` for the "shared" sound channel.

IMPORTANT: The Sound Manager only supports one channel of sound synthesis: You must pass `NULL` as the value of *`channel`*.

→ *`cmd`*
: One of the SndSmfCmdEnum values: either `SndSmfCmdPlay` (play the file) or `SndSmfCmdDuration` (return the duration of the file in milliseconds).

→ *`smfP`*
: The MIDI data; this can point to a `SndMidiRecHdrType` structure, or it can point directly to the actual MIDI data bytes in memory.

→ *`selP`*
: A pointer to a `SndSmfOptionsType` structure that defines performance parameters, such as volume, starting offset, and interruption tolerance. For default behavior, pass `NULL`. For more information, including default settings, see `SndSmfOptionsType`.

→ *`chanRangeP`*
: `A` pointer to a `SndSmfChanRangeType` structure that specifies the range of MIDI channels (in the SMF data) to use during playback. To play all channels, pass `NULL`.

→ *`callbacksP`*
: `A` pointer to a `SndSmfCallbacksType` structure that holds your callback functions. Pass `NULL` if you don't want any callbacks.

→ *`bNoWait`*
: This value is ignored. This function always finishes playing the SMF selection before returning (but see the Comments section, below).

Returns `errNone` if the operation completed successfully, or one of the following if an error occurs:

`sndErrBadParam`
: Invalid value passed to this function.

`sndErrBadChannel`
: Invalid sound channel.

`sndErrMemory`
: Insufficient memory.

`sndErrOpen`
: Tried to open channel that's already open.

`sndErrQFull`
: Can't accept more notes.

`sndErrFormat`
: Unsupported data format.

`sndErrBadStream`
: Invalid data stream.

`sndErrInterrupted`
: Play was interrupted.

Comments Although this call is always synchronous, you can register a "blocking" function that's called periodically as the MIDI file is playing. See `SndBlockingFuncType()` for more information.

Normally, playback is halted by events generated by user interaction with the screen, digitizer, or hardware-based buttons. You can override this behavior by setting the `interruptible` field of the *`selP`* parameter to `false`.

This function waits until any currently-playing simple sound has finished before starting playback of the requested MIDI data.

SndPlaySmfResource Function

Purpose Plays a MIDI track read out of an open resource database.

Declared In `SoundMgr.h`

Prototype

```
status_t SndPlaySmfResource (uint32_t resType,
    DmOpenRef dbRef, int16_t resID,
    SystemPreferencesChoice volumeSelector)
```

Parameters

→ *`resType`*
: SMF resource type.

→ *`dbRef`*
: Pointer to an open database. You can pass 0 to search all open resource databases for the specified resource type and ID.

→ *resID*
SMF resource ID.

→ *volumeSelector*
Volume setting; one of `prefSysSoundVolume`, `prefGameSoundVolume`, or `prefAlarmSoundVolume` (all defined in `Preferences.h`).

Returns `errNone` if the track was played successfully, or one of the following if an error occurs:

`sndErrBadParam`
The *volumeSelector* parameter is invalid or the SMF resource has invalid data.

`dmErrCantFind`
The specified resource doesn't exist.

other values
See `SndPlaySmf()`.

Comments This function plays the entire MIDI file using all MIDI channels. Playback is interrupted by a key down or digitizer event. No callbacks are specified.

This function waits until any currently playing simple sound has finished before starting playback of the requested MIDI data.

SndPlaySystemSound Function

Purpose Plays a pre-defined (simple) system sound.

Declared In `SoundMgr.h`

Prototype `void SndPlaySystemSound (SndSysBeepType` *beepID*`)`

Parameters → *beepID*
One of the system beep sound constants defined in the `SndSysBeepType` enum.

Returns Nothing.

Comments If you're playing an alarm (`sndAlarm`), the user's alarm volume preference setting is used. For all other system sounds, the system volume preference is used.

Alarm sounds (`sndAlarm`) are played synchronously: `SndPlaySystemSound()` blocks until the sound has been played. All other sounds are played asynchronously.

SndSetDefaultVolume Function

Purpose Sets the default sound volume levels cached by the Sound Manager.

Declared In `SoundMgr.h`

Prototype

```
void SndSetDefaultVolume (uint16_t *alarmAmpP,
   uint16_t *sysAmpP, uint16_t *defAmpP)
```

Parameters

→ *alarmAmpP*
Pointer to the alarm amplitude.

→ *sysAmpP*
Pointer to the system sound amplitude.

→ *defAmpP*
Pointer to the default amplitude for other sounds.

Returns Nothing.

Comments Any of the parameters may be `NULL`. In that case, the corresponding setting is not altered.

NOTE: It is usually not appropriate for an application to be setting the default sound volume levels. Accordingly, this function is rarely used by applications.

See Also `SndGetDefaultVolume()`

SndStreamCreate Function

Purpose Creates a new audio data stream that can be used to record or play back uncompressed, sampled audio data.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamCreate (SndStreamRef *channel,
   SndStreamMode mode, uint32_t samplerate,
   SndSampleType type, SndStreamWidth width,
   SndStreamBufferCallback func, void *userdata,
   uint32_t buffsize)
```

Parameters

← *`channel`*
: Token that represents the newly created stream.

→ *`mode`*
: One of the `SndStreamMode` constants that represents the “direction” of the data stream. Either `sndInput` (for recording), or `sndOutput` (for playback).

→ *`samplerate`*
: Sampling rate, in frames-per-second. Specify the native rate of the data, such as 22050, 44100, or 48000.

→ *`type`*
: Sample quantization and endianness (but see the section on “Data Formats,” below). Supply one of the values documented under “audio_type_t” on page 17.

→ *`width`*
: One of the constants documented under “SndStreamWidth” on page 21 that represents the number of channels of data in the stream.

→ *`func`*
: A callback function that gets called when another buffer is needed. See `SndStreamBufferCallback()` for a description of the callback function that you must implement.

→ *`userdata`*
: Caller-defined data that is passed to the callback function.

→ *`buffsize`*
: Preferred size (in frames) for the buffers that are passed to the callback function, *`func`*. Note that the actual buffer size (as

allocated by the Sound Manager) may be different from this request.

Returns `errNone` if the operation completed successfully, or one of the following if an error occurs:

`sndErrBadParam`
: *`channel`* is invalid, *`func`* is `NULL`, the sampling rate is too high (greater than 96000), or the device doesn't support some other specified sound parameter value.

`sndErrMemory`
: All streams are being used (there is a maximum of 16), or memory for this stream couldn't otherwise be allocated.

other errors
: The device couldn't allocate system resources for the stream.

Comments This function creates a new audio stream into which you can write (playback) or from which you can read (record) buffers of uncompressed, sampled audio data. The stream's "direction"—whether it will be used for recording or playback—is described by the *`mode`* argument.

You can create one input stream and as many as 15 output streams. The "active" end of a stream is hardwired to read from or write to the device's sound driver. This means you can't "redirect" an input stream to read from a file (for example), nor can you connect one output stream to another output stream in an attempt to create a filter chain. You can, however, collect data from the input stream, manipulate it, and then write it to an output stream.

Data Formats

The format of the data that flows through the stream is described by the *`sampleRate`*, *`type`*, and *`width`* arguments. If you're using an "extended" stream (see `SndStreamCreateExtended()`), you can also declare the data's encoding.

If you look at the audio_type_t constants, you'll see four flavors for each quantization type: a big-endian version, a little-endian version, a native-endian version (defined as one of the other two), and an "opposite" version, which has endianness opposite that of the native version. In general, you should use the native-endian version when choosing a value for the *`type`* parameter.

Running the Stream

The new stream starts running when you pass the *`channel`* token returned by this function to the `SndStreamStart()` function. This initiates a series of calls to your callback function (the *`func`* parameter), which is where the action is: Each callback invocation is passed a buffer into which you write or from which you read a chunk of audio data. The callback function is also passed the *`userdata`* parameter that you supply here. See `SndStreamBufferCallback()` for more information on the callback function.

Buffering and Latency

Currently, audio streams are double-buffered. With regard to playback, this means that while one buffer (buffer A) is being played, your callback function is placing data in the other buffer (B). When A is "empty," the Sound Manager seamlessly starts playing buffer B, and passes buffer A back to your callback; when B is empty, it starts playing A, and passes back B, and so on. It's important that your callback function fills the data buffers as quickly as possible—certainly no longer than it takes to play a buffer of data. This same double-buffer scheme is also applied to sound recording although, of course, for recording you're emptying each buffer (and doing something with the data) in your callback function.

Regarding latency, you can use the *`buffsize`* argument to suggest a buffer size and thereby increase or decrease latency, but you can't change the number of buffers. Keep in mind that the actual buffer size that's used may not be the same as the size you suggest; hardware and memory limitations may enforce a maximum or minimum buffer size. Also keep in mind that the buffer size is measured in frames (not bytes).

See Also `SndStreamStart()`, `SndStreamDelete()`, `SndStreamBufferCallback()`

SndStreamCreateExtended Function

Purpose Creates a new audio data stream that can be used to record or play back audio data.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamCreateExtended
   (SndStreamRef *channel, SndStreamMode mode,
   SndFormatType format, uint32_t samplerate,
   SndSampleType type, SndStreamWidth width,
   SndStreamVariableBufferCallback func,
   void *userdata, uint32_t buffsize)
```

Parameters

← *`channel`*
: Token that represents the newly created stream.

→ *`mode`*
: One of the `SndStreamMode` constants that represents the "direction" of the data stream. Either `sndInput` (for recording), or `sndOutput` (for playback).

→ *`format`*
: Constant that represents the encoding format of the data that you propose to pour through the stream. See "SndFormatType" on page 19, for a list of eligible values. Currently, only `sndFormatPCM` is supported.

→ *`sampleRate`*
: Sampling rate, in frames-per-second. Specify the native rate of the data, such as 22050, 44100, or 48000. The maximum rate is 96000.

→ *`type`*
: Sample quantization and endianness (see "Data Formats" on page 37 for advice on choosing this value). Supply one of the values documented under "audio_type_t" on page 17.

→ *`width`*
: One of the constants documented under "SndStreamWidth" on page 21 that represents the number of channels of data in the stream.

→ *`func`*
: A callback function that gets called when another buffer of data is needed. As implied by the name of the data type, the function accepts variable-sized buffers. See `SndStreamVariableBufferCallback()` for a

description of the callback function that you must implement.

→ *`userdata`*
Caller-defined data that is passed to the callback function.

→ *`buffsize`*
Preferred size (in frames) for the buffers that are passed to the callback function, *`func`*. Note that the actual buffer size (as allocated by the Sound Manager) may be different from this request.

Returns `errNone` if the operation completed successfully, or one of the following if there an error occurs:

`sndErrBadParam`
`channel` is invalid, *`func`* is `NULL`, the sampling rate is too high (greater than 96000), or the device doesn't support some other specified sound parameter value.

`sndErrMemory`
All streams are being used (there is a maximum of 16), or memory for this stream couldn't otherwise be allocated.

other errors
The device couldn't allocate system resources for the stream.

Comments With a few minor exceptions, this function is equivalent to `SndStreamCreate()`; see that function's Comments section for a description of how the stream creation functions generally work.

One difference between standard and extended streams: If you're using an extended stream, you can also declare the data's encoding. When selecting that encoding, be aware of the following:

- The data format that you specify for an input stream must match the data that's produced by the audio hardware.
- For an output stream, you can specify any of the formats that the Sound Manager supports; the data is automatically converted to the output hardware's native audio format. Whether your stream's format setting actually affects the hardware is undefined. For example, if you set an output stream to use a 48k sampling rate, that doesn't mean that the DAC will be set to 48k.
- Currently, only `sndFormatPCM` is supported.

Extended streams also allow for variable-sized buffers, as opposed to the fixed-sized buffers used by `SndStreamCreate()`.This enables support for variable-length encoded data (such as MP3). To accommodate the variable-sized buffer, the callback function's prototype changes slightly for an extended stream: see `SndStreamVariableBufferCallback()` for a full description of the callback function you use with `SndStreamCreateExtended()`.

SndStreamDelete Function

Purpose Stops the stream and deletes it.

Declared In `SoundMgr.h`

Prototype `status_t SndStreamDelete (SndStreamRef channel)`

Parameters → *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

Returns `errNone` if the operation completed successfully. Returns `sndErrBadParam` if the *`channel`* argument is invalid.

Comments `SndStreamDelete()` calls `SndStreamStop()` before deleting the stream. You should never call `SndStreamDelete()` from within your callback function.

SndStreamGetPan Function

Purpose Retrieves a stream's stereo balance.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamGetPan (SndStreamRef channel,
   int32_t *panposition)
```

Parameters → *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

← *`panposition`*
Pan value in the range [-1024 (extreme left), 1024 (extreme right)]. Center balance is 0.

Returns `errNone` if the operation completed successfully. Returns `sndErrBadParam` if *`channel`* is invalid or *`panposition`* is `NULL`.

See Also `SndStreamSetPan()`

SndStreamGetVolume Function

Purpose Retrieves the amplitude scalar for a sound stream.

Declared In `SoundMgr.h`

Prototype

```
status_t SndStreamGetVolume
   (SndStreamRef channel, int32_t *volume)
```

Parameters

→ *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

← *`volume`*
Amplitude scalar, in the range [0, 32k]. See `SndStreamSetVolume()` for more information.

Returns `errNone` if the operation completed successfully. Returns `sndErrBadParam` if *`channel`* is invalid or *`volume`* is `NULL`.

See Also `SndStreamSetVolume()`

SndStreamPause Function

Purpose Pauses or resumes a sample stream.

Declared In `SoundMgr.h`

Prototype

```
status_t SndStreamPause (SndStreamRef channel,
   Boolean pause)
```

Parameters

→ *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

→ *`pause`*
If `true`, the function pauses the stream; if `false`, it resumes the stream

Returns `errNone` if the operation completed successfully (which includes the situation where the stream is already in the requested state). Returns `sndErrBadParam` if *`channel`* is invalid.

Comments Currently, `SndStreamPause()` simply calls `SndStreamStop()` (if `pause` is `true`) or `SndStreamStart()` (if `pause` is `false`). See those functions for details about "pausing" and "resuming" a sound stream.

You can't nest pauses; a single resume request is effective, regardless of the number of times the stream has been told to pause.

SndStreamSetPan Function

Purpose Sets a stream's stereo balance.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamSetPan (SndStreamRef channel,
   int32_t panposition)
```

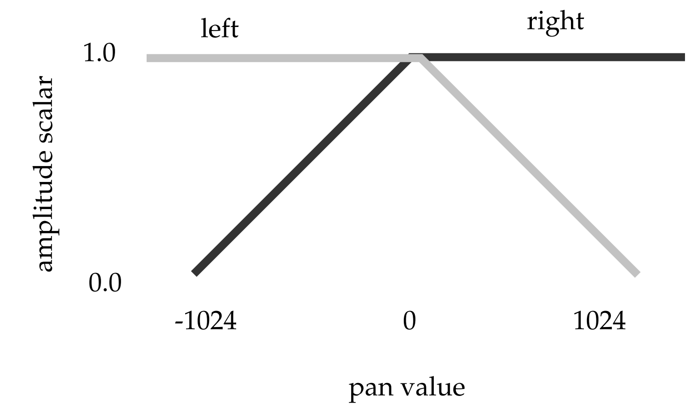
Parameters

→ *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

→ *`panposition`*
Pan value in the range [-1024 (full left), 1024 (full right)]. Center balance is 0. As a convenience, you can use the values described in "Stereo Pan Constants" on page 24." Note that values outside of the valid range may yield unexpected results (but don't generate an error).

Returns `errNone` if the operation completed successfully. Returns `sndErrBadParam` if *`channel`* is invalid.

Comments The pan value is used as a scalar on a channel's volume such that a channel increases from 0 (inaudible) to full volume as the pan value moves from an extreme to 0. Graphically, it looks like this:



See Also `SndStreamGetPan()`

SndStreamSetVolume Function

Purpose Sets the amplitude scalar for a sound stream.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamSetVolume
   (SndStreamRef channel, int32_t volume)
```

Parameters

→ *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

→ *`volume`*
Amplitude scalar in the range [0, 32k]. Values less than 0 are converted to 1024 (unity gain).

Returns `errNone` if the operation completed successfully. Returns `sndErrBadParam` if *`channel`* is invalid.

Comments The *`volume`* value is applied as an amplitude scalar on the samples that this stream's callback function produces. The scalar is in the range [0, 32k], where 1024 is unity gain (that is, the samples are multiplied by 1.0). The mapping of *`volume`* to a scalar is linear; thus a *`volume`* of 512 scales the samples by ~0.5, and 2048 scales by ~2.0, and so on.

To specify a user preference volume setting, supply one of the constants documented under "Volume Constants" on page 25. These values are guaranteed to be less than unity gain.

If the stream is stereo, both channels are scaled by the same amplitude scalar. To adjust the balance between the channels, use `SndStreamSetPan()`.

See Also `SndStreamGetVolume()`

SndStreamStart Function

Purpose Starts a sample stream running.

Declared In `SoundMgr.h`

Prototype `status_t SndStreamStart (SndStreamRef` *`channel`*`)`

Parameters → *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

Returns `errNone` if the operation completed successfully (`errNone` is returned even if the stream is already running). Returns `sndErrBadParam` if *`channel`* is invalid.

Comments If the stream is already running, `SndStreamStart()` returns immediately (with `errNone`). If it isn't running, the function starts the stream by initiating invocations of its callback function. If the stream is paused (through `SndStreamPause()`), the stream is resumed.

You can call this function from within another stream's callback function. This allows one stream to tell another stream to start playing.

See Also `SndStreamStop()`

SndStreamStop Function

Purpose Stops a sample stream from running.

Declared In `SoundMgr.h`

Prototype
```
status_t SndStreamStop (SndStreamRef channel)
```

Parameters → *`channel`*
Stream token, as returned from `SndStreamCreate()` or `SndStreamCreateExtended()`.

Returns `errNone` if the operation completed successfully (`errNone` is returned even if the stream is already stopped). Returns `sndErrBadParam` if *`channel`* is invalid.

Comments Stops a running sound stream by neglecting to call the stream's callback function. The stream remains in this suspended state until you call `SndStreamStart()`.

You can call this function from the stream's own callback function. In other words, a stream can stop itself.

Application-Defined Functions

SndBlockingFuncType Function

Purpose A callback function that is invoked periodically during SMF playback.

Declared In `SoundMgr.h`

Prototype
```
Boolean SndBlockingFuncType (void *chanP,
   uint32_t dwUserData,
   int32_t sysTicksAvailable)
typedef SndBlockingFuncType *SndBlockingFuncPtr
```

Parameters → *`chanP`*
A pointer to the sound channel on which the file is being played. Currently always `NULL`.

→ *`dwUserData`*
Application-defined data that's specified when the callback function is registered.

→ *`sysTicksAvailable`*
The amount of time, in milliseconds, available for completion of this function.

Returns Return `true` from your callback function if playback is to continue. Return `false` if playback is to be aborted.

Comments Your application's blocking callback is called whenever the MIDI parser is "between notes." Your application can do whatever it wants during this period, as long as it doesn't take more than *`sysTicksAvailable`* milliseconds.

Specify your blocking callback function using the `blocking` field of the `SndSmfCallbacksType` structure that you pass to `SndPlaySmf()`. Note that the `blocking` field is a `SndCallbackInfoType` structure; it contains a pointer to your callback function and a 32-bit value that is passed, as-is, to your callback.

See Also `SndComplFuncType()`

SndComplFuncType Function

Purpose A callback function that is invoked immediately after a MIDI file (SMF) finishes playing.

Declared In `SoundMgr.h`

Prototype
```
void SndComplFuncType (void *chanP,
   uint32_t dwUserData)
typedef SndComplFuncType *SndComplFuncPtr
```

Parameters

→ *`chanP`*
A pointer to the sound channel on which the file was playing. Currently always `NULL`.

→ *`dwUserData`*
Application-defined data that's specified when the callback function is registered.

Returns Return nothing.

Comments Specify your blocking callback function using the `completion` field of the `SndSmfCallbacksType` structure that you pass to `SndPlaySmf()`. Note that the `completion` field is a `SndCallbackInfoType` structure; it contains a pointer to your callback function and a 32-bit value that is passed, as-is, to your callback.

See Also `SndBlockingFuncType()`

SndStreamBufferCallback Function

Purpose In input mode, delivers a data buffer to your application. In output mode, allows your application to supply the next buffer's worth of data.

Declared In `SoundMgr.h`

Prototype

```
status_t (*SndStreamBufferCallback)
   (void *userdata, SndStreamRef channel,
   void *buffer, uint32_t numberofframes)
```

Parameters

→ *`userdata`*
Caller-defined data, as provided in the *`userdata`* parameter to `SndStreamCreate()`.

→ *`channel`*
Token that represents the stream to which this buffer belongs.

→ *`buffer`*
The data buffer.

→ *`numberofframes`*
Number of sample frames the buffer contains.

Returns Currently, the return value is ignored.

Comments The `SndStreamBufferCallback()` function that you create is invoked in input (recording) mode when the Sound Manager wants to deliver a new buffer of sound data to your application. In output (playback) mode, it is invoked when the Sound Manager needs another buffer of sound data. You associate your callback function with a given stream when you call `SndStreamCreate()`.

In input mode, your callback function should read the data from the data buffer. In output mode, your callback function should write data into the data buffer (and it must fill the entire buffer with data). In either case, you want to do this as quickly as possible to avoid data underflow.

Note that the arguments passed to your callback function tell you nothing about the format of the data. You can use the `userdata` argument to pass that information into the function.

See Also `SndStreamVariableBufferCallback()`

SndStreamVariableBufferCallback Function

Purpose In input mode, delivers a variable-length data buffer to your application. In output mode, allows your application to supply the next buffer's worth of data.

Declared In `SoundMgr.h`

Prototype

```
status_t (*SndStreamVariableBufferCallback)
   (void *userdata, SndStreamRef channel,
   void *buffer, uint32_t *bufferSizeP)
```

Parameters

→ *`userdata`*
Caller-defined data, as provided in the `userdata` parameter to `SndStreamCreateExtended()`.

→ *`channel`*
Token that represents the stream that this buffer belongs to.

→ *`buffer`*
The data buffer.

↔ *`bufferSizeP`*
Size of the buffer, in bytes.

Returns Currently, the return value is ignored.

Comments The `SndStreamVariableBufferCallback()` function that you create is invoked in input (recording) mode when the Sound Manager wants to deliver a new buffer of sound data to your application. In output (playback) mode, it is invoked when the Sound Manager needs another buffer of sound data. You associate your callback function with a given stream when you call `SndStreamCreateExtended()`.

In input mode, your callback function should read the data from the data buffer. In output mode, your callback function should write data into the data buffer and then reset the value in `bufferSizeP` to the amount of data that was actually written. Unlike `SndStreamBufferCallback()`, your callback function is not required to fill the entire buffer with data. Moreover, the data that it writes to the buffer doesn't have to meet any other threshold or requirement—for example, the buffer doesn't have to represent a certain amount of playback time. This flexibility is provided in order to support variable-length encoded data (such as MP3).

Whether your callback is reading from the buffer or writing to it, it should do this as quickly as possible to avoid data underflow.

Note that the arguments passed to your callback function tell you nothing about the format of the data. You can use the `userdata` argument to pass this information into the function.

Part II
Multimedia Library

The Multimedia Library controls the playback and recording of audio-visual media on Palm OS® devices.

3

Multimedia Applications

The Multimedia Library is an application-level API used to control the playback and recording of audio-visual media on Palm OS® devices. It provides a standard means for applications to reference media content stored locally on the device, stored on a network, or accessible from some attached hardware device such as a microphone or camera. The Multimedia Library also provides a means for applications to query and configure codecs and devices.

Multiple concurrent playback and recording sessions may be configured, and the processing of playback or recording sessions may continue in the background as the user uses other applications. Multiple components may interact with a session simultaneously.

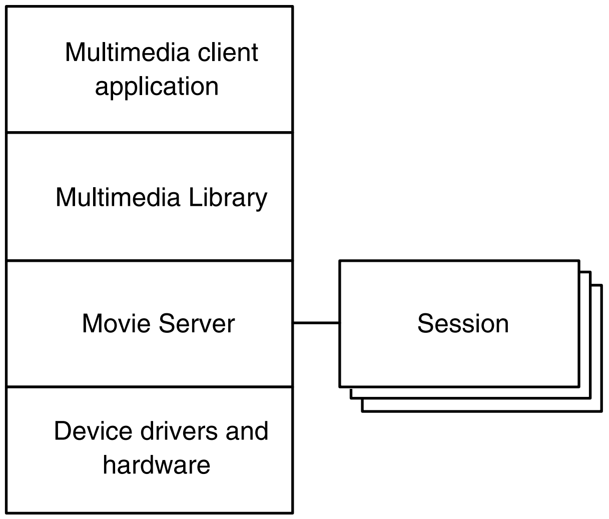
The Multimedia Library does not provide a means for developers to write file format handlers or codecs.

Overview

This section provides an architectural overview of the Multimedia subsystem.

Figure 3.1 shows the portions of the system that bring multimedia to the device.

Figure 3.1 Multimedia architecture



- The **multimedia client application** is an application that a third-party developer may write; it runs in the Application process. The Media Player is an example of such an application.
- The **Multimedia Library** provides the public APIs that multimedia clients use to access multimedia features.
- The **Movie Server** runs in the System process and provides all multimedia functionality. It spawns **sessions** that usually run in the Background process. Applications can control sessions via Multimedia Library function calls.
- Device drivers enable the use of specific multimedia hardware components such as cameras, microphones, speakers, and so forth.

The multimedia subsystem consists of several different kinds of objects that are described in the following sections.

Sessions

A **multimedia session** represents a recording or playback request, and controls the data transport. To use the Multimedia Library for playback, for example, you create a session, tell it where to find the data to play back, and then tell the session to start playing. The session automatically sends the data to the appropriate media output device such as the screen and/or the audio mixer.

For more information about using sessions, see "Working with Sessions" on page 59.

Sources

A **source** represents a source device. For a recording session, the source device might be a camera or a microphone. For a playback session, the source device might be a file or a network stream.

A source contains one or more **streams** of data from the source device. The stream defines the media format produced by the source device and is used to connect with the track. You connect each stream to a different track so that each stream's data goes to a different destination.

Destinations

A **destination** represents a destination device. For a recording session, the destination device might be a file or network stream. For a playback session, the destination might be the screen or the speakers.

Like sources, destinations contain one or more streams that specify the media format the destination expects, and they connect to a track.

Streams

A stream object represents multimedia data of a particular format. Each source and destination has at least one stream, which represents a single kind of data it produces or consumes. For example, a movie file source might have an audio stream and a video stream.

A stream object is created when you finalize a source or destination. You can enumerate the streams in sources and destinations by calling `MMSourceEnumerateStreams()` and `MMDestEnumerateStreams()`.

Tracks

A **track** represents a route for one type of media data from a source device to a destination device. For example, to play a movie would require two tracks: an audio track and a video track.

Tracks are responsible for encoding or decoding data as it is recorded or played.

Tracks sometimes use a **track callback filter**, which is usually supplied by the application. A multimedia application can set a track callback filter if it wants to handle the data itself. If so, the track ensures that data is passed from the source device to the track filter, which forwards it to the callback function and then passes it on to the destination.

To register a track callback filter, call `MMTrackInsertCallbackFilter()`.

Codecs

In order to play from and capture to encoded media files, the Multimedia Library uses software components called **codecs** (an abbreviation for encoder/decoder.) A codec translates media data from one format to another. Each codec supports a particular encoding algorithm, such as MP3 or MS-ADPCM.

A stream uses a codec during playback. During recording, the stream uses an encoder before writing to the file stream.

Applications can enumerate the available codecs with `MMCodecClassEnumerate()`, and can enumerate the available file formats with `MMFileFormatEnumerate()`. File formats are distinct from codecs because a file format may encapsulate many kinds of encoded data.

File formats are described by an `MMFormatType` value. Codecs are represented by an `MMCodecClassID`, and the `MMPropertyGet()` function may be used to obtain more information about a given codec (such as name, creator, and source or destination format.) All codec properties are read-only.

PalmSource provides several built-in codecs:

- MS-ADPCM Audio Decoder

- DVI-ADPCM Audio Decoder
- MPEG Audio Layer I/II Decoder
- MPEG-1 Video Decoder
- MS-ADPCM Audio Encoder
- AVI Extractor
- MPEG Audio Extractor
- MPEG-1 Extractor
- WAV Extractor
- WAV Composer

Different codecs, such as MP3 and MPEG-4, might be available depending on the device manufacturer.

Formats

Formats are used to specify what multimedia formats an object can work with.

An object that can handle both audio and video data typically has two format objects: one specifying its audio constraints and one specifying its video constraints. Format objects themselves are made up of key/value pairs specifying one value for one attribute. For example, a raw audio format has keys for sample type, frame rate, channel count, and so on.

A format object is referenced by an `MMFormat`. Formats may be retrieved as property values by `MMPropertyGet()` or returned by `MMFormatCreate()`, and in every case must be explicitly deleted by the application when it has finished using them. An `MMFormat` has a type, `MMFormatType`, which describes the basic kind of media such as `P_FORMAT_RAW_AUDIO`, `P_FORMAT_MPEG4_VIDEO`, etc. Every format type has an associated set of format keys, which are documented in `MMFormatDefs.h`. See Chapter 6, "Multimedia Formats," for a list of the keys and values that formats use.

During format negotiation, two formats are inspected to see if they are compatible, and any wild values are replaced with actual values.

A format object stores a value for each format key. Values are typed by an `MMTypeCode`, allowing formats to contain a wide range of data. Some common value types are `P_MM_INT32_TYPE`, `P_MM_BOOL_TYPE`, and `P_MM_WILD_TYPE`. The last type simply means that any value is acceptable for the key; the library assigns that key an appropriate value when a session is finalized, and no data is stored.

Property Sets

Property sets expose configurable parameters, or properties, which control an object's behavior. For example, an audio-renderer supports the `P_MM_TRACK_PROP_VOLUME` property to allow control of output volume.

A property consists of a key/value pair, where the key is a 32-bit constant identifier and the value is a typed chunk of data with a given size. Property values are generally either 32-bit integers or character strings, though some properties may have more complex values (such as those that represent dimensions, regions, or media formats).

All objects that support properties are identified by 32-bit ID values, which identify both the object instance and its type.

For more information on properties, see "Working with Properties" on page 60.

Using the Multimedia Library

The Multimedia Library is a shared library that the system automatically loads when needed and unloads when not needed. You don't need to do anything to load or initialize the library.

This section covers these topics:

- Working with Sessions
- Working with Properties
- Working with Enumerations
- Working with the URL Scheme

Working with Sessions

A session provides a context for an application's media playback or recording tasks. Before using the Multimedia Library to play or record, you must create a session by calling `MMSessionCreate()`. In this function, you specify the session class, which indicates if the session is for playback or recording (capture).

A session is described in terms of sources, destinations, streams, and tracks. Sources and destinations represent the files, network streams, and devices used to get multimedia data into and out of the media-processing engine. Each source and destination has at least one stream, which represents a single kind of data it produces or consumes. For example, a movie file source might have an audio stream and a video stream.

A track represents a route for media data in the session. Tracks take data from a source stream, apply some processing (such as decoding and filtering), and send it to a destination stream.

To configure a session, your application must first add whichever sources and destinations it requires by calling `MMSessionAddSource()` and `MMSessionAddDest()`. Then finalize the sources and destinations by calling `MMSourceFinalize()` and `MMDestFinalize()` (if there are destinations). Then, enumerate the available streams by calling `MMSourceEnumerateStreams()` and `MMDestEnumerateStreams()`.

Finally, add tracks for the streams you wish to play back or capture. You can add tracks manually one at a time with `MMSessionAddTrack()`, or automatically by using `MMSessionAddDefaultTracks()`. When you have added all the desired tracks to the session, call `MMSessionFinalize()` to prepare it for performance. After calling this function, no new sources, destinations, or tracks may be added.

To reconfigure a session after it has been finalized, you must first call `MMSessionRemoveAll()`.

To start, stop, pause, or otherwise control the playback or capture process, call `MMSessionControl()`. This call exposes the various transport operations such as run, pause, stop, prefetch, grab a still image, and refresh the display.

For playback sessions you can also call `MMSessionSeek()`, which instructs the decoder to jump to a new position in the content stream. Note that seek functionality may not always be available, depending on the kind of content and on the location it's being streamed from.

You can determine the current state of the session by calling `MMSessionGetState()`.

If you want your application to receive multimedia event notifications from a session, call `MMSessionRegisterCallback()` to register a callback function. If you want your application to receive multimedia event notifications of a persistent session even when the application is no longer running, call `MMSessionRegisterLaunch()`. Then, when session events occur, the application is sublaunched with the launch code `sysAppLaunchCmdMultimediaEvent`.

Working with Properties

All objects that support properties are identified by 32-bit ID values, which identify both the object instance and its type. This allows a single set of property functions to operate on any kind of property-bearing object. There are four property functions:

- `MMPropertySet()`: sets a property value
- `MMPropertyGet()`: returns a property value
- `MMPropertyInfo()`: returns various information about a property, such as its minimum, maximum, and default values, whether it is readable and/or writable, and its type
- `MMPropertyEnumerate()`: lists the potential values of a property

Not all of these operations are applicable to all properties, or to all entities that have associated properties; for example, `MMPropertyEnumerate()` may return `sysErrNotAllowed` if the specified `MMPropInfoType` parameter does not apply. For example, some properties, such as `P_MM_TRACK_PROP_VOLUME`, have continuous values with a well-defined minimum and maximum, but have such a wide range of possible values that it doesn't make sense for `MMPropertyEnumerate()` to list them. These are referred to as "continuous-valued" properties. Other properties, such as P_MM_TRACK_PROP_CODEC_CLASS, allow a discrete set of values for which the concepts of minimum and maximum do not apply, and are thus termed "discrete-valued" properties.

Here's an example of retrieving the current value of a property by calling MMPropertyGet() and passing the key:

```
err = MMPropertyGet(session, P_MM_CONTENT_PROP_DURATION,
P_MM_INT64_TYPE, &longduration, 0);
```

Here's an example of setting the value of a property by calling MMPropertySet() and passing both the key and the value. The value must have the same type as the current value of the property:

```
err = MMPropertySet(session, P_MM_SESSION_PROP_PLAYBACK_RATE,
P_MM_INT32_TYPE, &rate, 0);
```

Working with Enumerations

There are several cases in the Multimedia Library where a given component or object provides access to a set of values or references to objects. A common iterator-based enumeration scheme is used in each case. For example, here is the file format enumerator function:

```
status_t MMFileFormatEnumerate
(int32_t *ioIterator, MMFormatType *outFormat)
```

The value pointed to by *ioIterator* must be treated as opaque by the caller, with two exceptions:

- Before the first call to the enumeration function, the value of *ioIterator* must be set to P_MM_ENUM_BEGIN.
- When the set has been exhausted, the enumeration function will set the value of *ioIterator* to P_MM_ENUM_END.

Other values are only guaranteed to be meaningful to the Multimedia Library. If an enumeration function is called with an invalid iterator, or an iterator value of P_MM_ENUM_END, it returns sysErrBadIndex.

An example of enumerating the tracks in a session is shown in Listing 3.1.

Listing 3.1 Enumerating tracks

```
status_t err;
MMTrackID outTrack;
int32_t ioIterator = P_MM_ENUM_BEGIN;
while(true)
{
  err = MMSessionEnumerateTracks(session, &ioIterator,
        &outTrack);
  if(err != errNone)
    break;
  // do something with each track returned in outTrack
}
```

Working with the URL Scheme

Files on expansion cards or other VFS volumes can be accessed using URLs with the following syntax:

```
FileURL = "file://" ["localhost"] "/" [VolumeLabel] "/" Path
```

where *`VolumeLabel`* can use any characters except for control codes (unprintable characters), "/", and "?"; and *`Path`* can use any characters except for control codes and "?".

You may use or omit the optional server name component of the URL ("localhost") when using the file scheme. When omitting it, you must still specify the following slash, so you would write the scheme like this: `file:///`...

instead of this: `file://locahost/`...

The volume label is case-sensitive. The case requirements of the rest of the URL follow the convention of the file system that this volume is mounted on.

If no volume label is specified, all volumes are searched for the given path.

Here's some URL examples:

`file://localhost/MySDCardVolumeName/PALM/Launcher/Giraffe.prc`

`file:///OtherCardVolumeName/PALM/Launcher/Giraffe.prc`

```
file:////PALM/Launcher/Giraffe.prc
```

```
file:////Audio/mySong.mp3
```

URL name conflicts are resolved as follows: All volumes matching the specified name are searched and if multiple databases sharing the same pathname are found, the most recent is returned. If multiple selections share the same date, the file-selection behavior is nondeterministic (this only happens if no volume label is specified).

URLs are also used to access specific devices (such as a camera or a microphone) using URLs with the scheme `palmdev://`. For example, the URL `palmdev:///Media/Default/VideoOut` specifies any video playback device. For more examples of `palmdev://` URLs, see "Default URLs" on page 119.

Example Playback Session

This section describes how to use the Multimedia Library to play multimedia content.

The basic steps involved in playback are listed here and described in more detail later:

1. "Creating the Session" on page 64.
2. "Adding Source Content" on page 64.
3. "Adding Tracks" on page 64.
4. "Finalizing the Session" on page 65.
5. "Playing the File" on page 65.

Creating the Session

The first basic step is to create the multimedia session:

1. Call the function `MMSessionCreate()`, passing `P_MM_SESSION_CLASS_DEFAULT_PLAYBACK` as the session class (to create a playback session).

   This tells the Movie Server to create the session. It publishes the following keys and values in the session property set:

   `P_MM_SESSION_PROP_PLAYBACK_RATE`: 1

   `P_MM_SESSION_PROP_MARKER`: –1

   `P_MM_SESSION_DEFAULT_AUDIO_ENABLE`: true

   `P_MM_SESSION_DEFAULT_VIDEO_ENABLE`: true

   `P_MM_SESSION_DEFAULT_SOURCE_RECT`: null

   `P_MM_SESSION_DEFAULT_DEST_RECT`: null

   `P_MM_SESSION_DEFAULT_AUDIO_VOLUME`: 1024

2. Optionally, call `MMSessionRegisterCallback()` to register a callback function that will receive event notifications from the session. This step is not required, but is quite useful.

Adding Source Content

After creating the session, you need to add source content:

1. Call `MMSessionAddSource()`, passing it the URL of the multimedia file to be opened.
2. Call `MMSourceFinalize()`, which prepares the media streams for use. One stream is created for each track in the content; for a movie, there is typically one track for audio and one track for video.

Adding Tracks

You now need to specify where the multimedia data should go. To do so, call `MMSessionAddDefaultTracks()`, passing the session ID (returned by `MMSessionCreate()`), the source ID (returned by `MMSessionAddSource()`), and the constant `P_MM_DEFAULT_DEST`.

Finalizing the Session

Now you need to finalize the session:

1. Call `MMSessionFinalize()`.
2. You may want to call `MMSessionReleaseOwnership()`. This transfers ownership of the session back to the Movie Server now that your application has finished creating it.

   When the Movie Server owns the session, if the application exits, the session can continue playing in the background.
3. For video playback, set a destination rectangle of the size required by the session so that it is ready to display the content. To do this, call `MMPropertySet()` to set a `P_MM_TRACK_PROP_DEST_RECT` property for the track, like this:

```
MMPropertySet(track, P_MM_TRACK_PROP_DEST_RECT, P_MM_RECT_TYPE, rect, 0);
```

   The *`rect`* parameter is the destination rectangle for the video content.

Playing the File

Now that the file has been opened and the objects that are required to play the file have been created, the application can return control to the user. The user presses the Play button on the device, specifying that playback should begin. (The Play button can be a hardware button or a button displayed by the application in its user interface.)

Call `MMSessionControl()` with the `P_MM_SESSION_CTL_RUN` control code. This function forwards the control code to the Movie Server session. It sets its state to `P_MM_SESSION_RUNNING` and sends an event to notify the application of its change in state.

You can call `MMSessionControl()` with other control codes to perform other functions such as pause and stop. To seek forward or backward in the session, for example, to implement fast forward and rewind functions, call `MMSessionSeek()`.

Video and audio playback occur in separate threads.

Example Recording Session

This section describes how to use the Multimedia Library to record audio content.

A recording session begins when the user presses the Record button in the application. The Record button can be a hardware button or a button displayed by the application in its user interface, or some other mechanism that you devise.

1. First, the application must create a name for the file that will hold the recorded data.
2. Then create the session by calling the function `MMSessionCreate()`, passing `P_MM_SESSION_CLASS_DEFAULT_CAPTURE` as the session class (to create a recording session).
3. Call `MMSessionRegisterCallback()` to register a callback function that will receive event notifications from the session.
4. Add the source by calling `MMSessionAddSource()`, specifying `P_MM_DEFAULT_AUDIO_CAPTURE_URL` as the URL. This source represents the means by which the device receives audio input, such as a microphone or line-input jack.
5. Call `MMSourceFinalize()`.
6. Obtain the source stream ID by enumerating the source streams (there should be only one) by calling `MMSourceEnumerateStreams()`.
7. Add the destination by calling `MMSessionAddDest()`.
8. Set the `P_MM_DEST_PROP_FILE_FORMAT` property of the destination to `P_FORMAT_WAV_STREAM` (or whatever audio format you want) by calling `MMPropertySet()`, like this:

```
int32_t streamFormat = P_FORMAT_WAV_STREAM;
MMPropertySet(dest, P_MM_DEST_PROP_FILE_FORMAT, P_MM_INT32_TYPE, &streamFormat,
0);
```

9. Finalize the destination by calling `MMDestFinalize()`.
10. Obtain the destination stream ID by enumerating the destination streams (there should be only one) by calling `MMDestEnumerateStreams()`.

11. Create a media format object and set its type to `P_FORMAT_MSADPCM_AUDIO`, like this:

```
MMFormat encoding = 0;
MMFormatCreate(&encoding);
MMFormatSetType(encoding, P_FORMAT_MSADPCM_AUDIO);
```

12. Add a track to provide a route from the audio source to the file destination by calling `MMSessionAddTrack()`, like this:

```
err = MMSessionAddTrack(session, sourceStream, 0, destStream, encoding,
&track);
```

    The source stream ID is returned by `MMSourceEnumerateStreams()` in step 6. The destination stream ID is returned by `MMDestEnumerateStreams()` in step 10.

13. Finalize the session by calling `MMSessionFinalize()`.

    Then the session's state is set to `P_MM_SESSION_READY` to signal that recording can begin.

14. To start recording, call `MMSessionControl()` with the `P_MM_SESSION_CTL_RUN` control code.

    This function forwards the control code to the Movie Server session. It sets its state to `P_MM_SESSION_RUNNING` and sends an event to notify the application of its change in state.

You can call `MMSessionControl()` with other control codes to perform other functions such as pause and stop. Pass the control code `P_MM_SESSION_CTL_STOP` to stop recording (and write header data to the file so that it can be used immediately.)

The session is deleted after the application that created it exits, unless it has called `MMSessionReleaseOwnership()` to allow the session to keep running in the background. If the application does call this function, then it should call `MMSessionDelete()` or `MMSessionAcquireOwnership()` to ensure that the session gets deleted.

4

Multimedia Library Definitions

This chapter describes the structures and types defined in the header file `MMDefs.h`:

Multimedia Definitions Structures and Types

MMCodecClassID Typedef

Purpose The class ID of a decoder or encoder object.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMCodecClassID`

MMDestID Typedef

Purpose The ID of a multimedia destination object.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMDestID`

MMEvent Typedef

Purpose	Not currently used.
Declared In	`MMDefs.h`
Prototype	`typedef int32_t MMEvent`

MMFilterID Typedef

Purpose	Not currently used.
Declared In	`MMDefs.h`
Prototype	`typedef int32_t MMFilterID`

MMSessionClassID Typedef

Purpose	The class ID of a session subclass.
Declared In	`MMDefs.h`
Prototype	`typedef int32_t MMSessionClassID`

MMSessionID Typedef

Purpose	The ID of a session object.
Declared In	`MMDefs.h`
Prototype	`typedef int32_t MMSessionID`

MMSourceID Typedef

Purpose	The ID of a multimedia source object.
Declared In	`MMDefs.h`
Prototype	`typedef int32_t MMSourceID`

MMStreamID Typedef

Purpose The ID of a stream object.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMStreamID`

MMTrackID Typedef

Purpose The ID of a track object.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMTrackID`

Multimedia Definitions Constants

Complex Property Values Enum

Purpose Specifies a more complex type of value stored in a property set.

Declared In `MMDefs.h`

Constants `P_MM_RECT_TYPE = MM_TYPE_CODE('Rct')`
A `RectangleType` structure.

`P_MM_FORMAT_TYPE = MM_TYPE_CODE('Fmt')`
A `MMFormat` object.

Enumerations Enum

Purpose Specifies the beginning and end of an enumeration.

Declared In `MMDefs.h`

Constants `P_MM_ENUM_BEGIN = 0`
Tells a function to begin the enumeration.

`P_MM_ENUM_END = -1`
Returned by a function when there are no more values to enumerate.

Miscellaneous Constants

Purpose Other constants defined in `MMDefs.h`.

Declared In `MMDefs.h`

Constants

`#define P_MM_INVALID_ID 0`
Specifies an invalid ID.

`#define P_MM_TYPE_CODE_MASK 0x7f7f7f00`
Mask of all `MMTypeCode` values.

`#define P_MM_TYPE_CODE_SHIFT 8`
Amount to shift to get `MMTypeCode` values.

MMPropInfoType Typedef

Purpose Specifies the type of data to retrieve from a property set.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMPropInfoType`

Constants

`P_MM_PROP_INFO_DEFAULT`
Obtain the default value.

`P_MM_PROP_INFO_MINIMUM`
Obtain the minimum value.

`P_MM_PROP_INFO_MAXIMUM`
Obtain the maximum value.

`P_MM_PROP_INFO_READABLE`
Returns whether the value is readable.

`P_MM_PROP_INFO_WRITABLE`
Returns whether the value is writable.

`P_MM_PROP_INFO_TYPE_CODE`
Returns the type of value stored for the property.

Comments The function `MMPropertyInfo()` uses these values to obtain specific information from a property set.

MMSeekOrigin Typedef

Purpose A multimedia application uses these constants to specify where to being a seek operation.

Declared In `MMDefs.h`

Prototype `typedef int8_t MMSeekOrigin`

Constants `P_MM_SEEK_ORIGIN_BEGIN`
Start at the beginning of the file.

`P_MM_SEEK_ORIGIN_CURRENT`
Start at the current location.

`P_MM_SEEK_ORIGIN_END`
Start at the end of the file.

See Also `MMSessionSeek()`

MMTypeCode Typedef

Purpose Specifies the type of value stored in a property or `MMFormat` object.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMTypeCode`

Constants `P_MM_UNDEFINED_TYPE = MM_TYPE_CODE(0)`
Not defined.

`P_MM_WILD_TYPE = MM_TYPE_CODE('wld')`
Wild.

`P_MM_RAW_TYPE = MM_TYPE_CODE('raw')`
Raw data.

`P_MM_INT8_TYPE = MM_TYPE_CODE('i08')`
8-bit integer.

`P_MM_INT16_TYPE = MM_TYPE_CODE('i16')`
16-bit integer.

`P_MM_INT32_TYPE = MM_TYPE_CODE('i32')`
32-bit integer.

`P_MM_INT64_TYPE = MM_TYPE_CODE('i64')`
64-bit integer.

`P_MM_BOOL_TYPE = MM_TYPE_CODE('bol')`
Boolean value.

`P_MM_STRING_TYPE = MM_TYPE_CODE('str')`
String value.

Object Property Key Bases Enum

Purpose Used to construct base values for the property keys used by different objects.

Declared In `MMDefs.h`

Constants

`P_MM_PROP_OBJECT_MASK = 0xFF000000L`
Mask of all object property key base values.

`P_MM_PROP_OBJECT_SESSION = (1L << 24)`
Identifies session object property keys.

`P_MM_PROP_OBJECT_CONTENT = (2L << 24)`
Identifies multimedia content property keys.

`P_MM_PROP_OBJECT_SOURCE = (3L << 24)`
Identifies source object property keys.

`P_MM_PROP_OBJECT_DEST = (4L << 24)`
Identifies destination object property keys

`P_MM_PROP_OBJECT_STREAM = (5L << 24)`
Identifies stream object property keys.

`P_MM_PROP_OBJECT_TRACK = (6L << 24)`
Identifies track object property keys.

`P_MM_PROP_OBJECT_DEVICE = (7L << 24)`
Not currently used.

`P_MM_PROP_OBJECT_SESSION_CLASS = (8L << 24)`
Identifies session subclass properties.

`P_MM_PROP_OBJECT_CODEC_CLASS = (9L << 24)`
Identifies codec object property keys.

Property Base Enum

Purpose Used to construct property key constants.

Declared In `MMDefs.h`

Constants

`P_MM_STANDARD_PROP_BASE = 0x00010000L`
Identifies PalmSource-defined property keys.

`P_MM_USER_PROP_BASE = 0x00020000L`
Identifies licensee-defined property keys.

`P_MM_PRIVATE_PROP_BASE = 0x00030000L`
Identifies private property keys.

Property Key Base Values Enum

Purpose Base values for property key constants used by various objects.

Declared In `MMDefs.h`

Constants

`P_MM_SESSION_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_SESSION`
Base value for session object property keys.

`P_MM_CONTENT_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_CONTENT`
Base value for content property keys.

`P_MM_SOURCE_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_SOURCE`
Base value for source object property keys.

`P_MM_DEST_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_DEST`
Base value for destination object property keys

`P_MM_STREAM_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_STREAM`
Base value for stream object property keys.

`P_MM_TRACK_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_TRACK`
Base value for track object property keys.

`P_MM_DEVICE_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_DEVICE`
: Not currently used.

`P_MM_SESSION_CLASS_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_SESSION_CLASS`
: Base value for session subclass properties.

`P_MM_CODEC_CLASS_PROP_BASE = P_MM_STANDARD_PROP_BASE | P_MM_PROP_OBJECT_CODEC_CLASS`
: Base value for codec object property keys.

Session Event Causes Enum

Purpose Values for the `MMSessionEvent` `eventCause` field.

Declared In `MMDefs.h`

Constants

`P_MM_EVENT_CAUSE_UNKNOWN = 0x01`
: The cause is unknown.

`P_MM_EVENT_CAUSE_REQUESTED_BY_APP = 0x02`
: The application requested that the event occur.

`P_MM_EVENT_CAUSE_END_OF_STREAM = 0x03`
: All tracks stopped because there is no more data to write or to read.

`P_MM_EVENT_CAUSE_INVALID_STREAM = 0x04`
: All tracks stopped because bad data was detected in the stream.

`P_MM_EVENT_CAUSE_STORAGE_FULL = 0x05`
: All tracks stopped because the destination storage is full.

`P_MM_EVENT_CAUSE_CUSTOM_BASE = 0x1000`
: Base value after which you may add your own custom events.

Session Notifications Enum

Purpose Notifications sent by the session when something occurs. These are used as values for the `MMSessionEvent` `eventCode` field.

Declared In `MMDefs.h`

Constants `P_MM_EVENT_SESSION_STATE_CHANGED = 0x01`
Sent to the client process when a session's state changes such as from ready to running to stopped.

`P_MM_EVENT_SESSION_MARKER_EXPIRED = 0x02`
Sent to the client process when a marker requested by the application has been reached on a particular track.

`P_MM_EVENT_SESSION_DELETING = 0x03`
The session is in the process of being deleted.

`P_MM_EVENT_SESSION_WARNING = 0x04`
A recoverable error has occurred during the session.

`P_MM_EVENT_CUSTOM_BASE = 0x1000`
Base value after which licensees may add their own custom events.

Multimedia Definitions Functions and Macros

MM_TYPE_CODE Macro

Purpose Used to construct `MMTypeCode` values.

Declared In `MMDefs.h`

Prototype `#define MM_TYPE_CODE (`*`code`*`)`

Parameters → *`code`*
An 8-bit number.

Returns One of the `MMTypeCode` values.

5
Multimedia Codecs

This chapter describes multimedia constants and functions related to codecs:

The header file `MMCodecClass.h` declares the API that this chapter describes.

Multimedia Codec Constants

Codec Class Properties

Purpose Defines property keys that can be used to access information in codec classes.

Declared In `MMCodecClass.h`

Constants

```
#define P_MM_CODEC_CLASS_PROP_CREATOR
  (P_MM_CODEC_CLASS_PROP_BASE | 0x0003L)
```
The codec class's creator ID.

```
#define P_MM_CODEC_CLASS_PROP_DEST_FORMAT
  (P_MM_CODEC_CLASS_PROP_BASE | 0x0005L)
```
The codec's destination format if it is an encoder.

```
#define P_MM_CODEC_CLASS_PROP_NAME
  (P_MM_CODEC_CLASS_PROP_BASE | 0x0001L)
```
The codec's name.

```
#define P_MM_CODEC_CLASS_PROP_SOURCE_FORMAT
  (P_MM_CODEC_CLASS_PROP_BASE | 0x0004L)
```
The codec's source format if it is a decoder.

```
#define P_MM_CODEC_CLASS_PROP_VERSION
  (P_MM_CODEC_CLASS_PROP_BASE | 0x0002L)
```
The codec's version number.

Multimedia Codec Functions and Macros

MMCodecClassEnumerate Function

Purpose Iterates through the available codecs, both encoders and decoders.

Declared In `MMCodecClass.h`

Prototype
```
status_t MMCodecClassEnumerate (MMFormatType type,
   int32_t *ioIterator,
   MMCodecClassID *outCodecClassID)
```

Parameters

→ *`type`*
: The type of codecs to enumerate. Specify one of the constants listed in "formatType" on page 93, or specify `P_FORMAT_UNKNOWN` to enumerate codecs of all format types.

↔ *`ioIterator`*
: Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outCodecClassID`*
: Pointer to the ID of the next available codec.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The iterator is invalid or the type doesn't match previous calls in the same iteration set.

`sysErrBadIndex`
: The iterator value is invalid or past the last item in the set.

MMFileFormatEnumerate Function

Purpose Iterates through the supported file formats (these are distinct from codecs because a file format may encapsulate many kinds of encoded data).

Declared In `MMCodecClass.h`

Prototype

```
status_t MMFileFormatEnumerate
    (int32_t *ioIterator, MMFormatType *outFormat)
```

Parameters

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outFormat`*
Pointer to the ID of the next available format.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The iterator is invalid or the type doesn't match previous calls in the same iteration set.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

6

Multimedia Formats

This chapter describes multimedia structures, types, constants, and functions related to formats. It covers:

The header files `MMFormatDefs.h` and `MMFormat.h` declare the API that this chapter describes.

Multimedia Format Structures and Types

MMFormat Typedef

Purpose Identifies a multimedia format object.

Declared In `MMFormat.h`

Prototype `typedef int32_t MMFormat`

MMFormatType Typedef

Purpose Identifies a multimedia format type.

Declared In `MMFormat.h`

Prototype `typedef int32_t MMFormatType`

Comments Format types are defined by the `formatType` enum.

Multimedia Format Constants

Format Key Constants

Purpose Values used as keys to format terms. The descriptions below explain what the value is for each key. Each value is a 32-bit integer unless otherwise specified.

Declared In `MMFormatDefs.h`

Constants

`#define P_FORMATKEY_BYTE_ORDER "byte_order"`
The byte order for raw audio. Possible values are `LITTLE_ENDIAN`, `BIG_ENDIAN`, or `HOST_ENDIAN`.

`#define P_FORMATKEY_RAW_AUDIO_TYPE "raw_audio_type"`
Base type used for raw audio. Possible values are `fmtRawAudioType`.

`#define P_FORMATKEY_RAW_AUDIO_BITS "raw_audio_bits"`
If `P_FORMATKEY_RAW_AUDIO_TYPE` is `P_AUDIO_INT32`, the value for this provides the actual number of bits used in a given 32-bit sample.

`#define P_FORMATKEY_CHANNEL_USAGE "channel_usage"`
Number of audio channels for a raw, ADPCM, or MPEG audio stream. Possible values are `fmtAudioChannelUsage`.

`#define P_FORMATKEY_ENCODED_BIT_RATE "enc_bit_rate"`
The audio or video bit rate for MPEG formats.

`#define P_FORMATKEY_FRAME_RATE "frame_rate"`
A floating-point value specifying the number of frames processed per second for an audio or video stream.

`#define P_FORMATKEY_BUFFER_FRAMES "buffer_frames"`
An integer specifying the number of frames per buffer for audio data.

`#define P_FORMATKEY_WIDTH "width"`
Width in native pixels of a video frame or graphics file.

`#define P_FORMATKEY_HEIGHT "height"`
Height in native pixels of a video frame or graphics file.

`#define P_FORMATKEY_BYTES_PER_ROW "bytes_per_row"`
: An integer specifying the number of bytes per row of a raw video frame or of a graphics file.

`#define P_FORMATKEY_PIXEL_FORMAT "pixel_format"`
: 64-bit integer describing the pixel format of a raw video frame or of a graphics file.

`#define P_FORMATKEY_VIDEO_ORIENTATION "video_orientation"`
: Video orientation for graphics still. Possible values are defined in `fmtVideoOrientation`.

`#define P_FORMATKEY_MSADPCM_BITS_PER_SAMPLE "msadpcm_sample_bits"`
: Number of bits per sample for a mono-channel MS-ADPCM sound stream.

`#define P_FORMATKEY_MSADPCM_COEFS "msadpcm_coefs"`
: Variable size 16-bit coefficient table in host-endian format.

`#define P_FORMATKEY_MPEG12_AUDIO_REVISION "mpeg12_audio_rev"`
: The specific version of MPEG audio. Possible values are defined in `fmtMPEG12AudioRevision`.

`#define P_FORMATKEY_MPEG12_AUDIO_LAYER "mpeg12_audio_layer"`
: MPEG-1 or MPEG-2 audio layer. Possible values are in `fmtMPEG12AudioLayer`.

`#define P_FORMATKEY_MPEG12_AUDIO_CHANNEL_MODE "mpeg12_audio_channel_mode"`
: MPEG-1 or MPEG-2 audio channel mode. Possible values are defined in `fmtMPEG12AudioChannelMode`.

`#define P_FORMATKEY_DVIADPCM_BITS_PER_SAMPLE "dviadpcm_sample_bits"`
: Number of bits per sample in a mono-channel Intel/DVI ADPCM audio file.

`#define P_FORMATKEY_MPEG4AUDIO_OBJECT_PROFILE "mpeg4audio_object_profile"`
: Type of audio object profile used for MPEG-4 audio. Possible values are defined in `fmtMPEG4AudioObjectProfile`.

`#define P_FORMATKEY_MPEG4AUDIO_TF_CODING "mpeg4audio_tf_coding"`
: MPEG-4 audio track time/frequency coding format. Possible values are defined in `fmtMPEG4AudioTFCoding`.

`#define P_FORMATKEY_MPEG4AUDIO_TF_FRAME_LENGTH "mpeg4audio_tf_frame_length"`
: Frame length for MPEG-4 audio time/frequency coding format.

`#define P_FORMATKEY_MPEG4AUDIO_TF_CORE_CODER_DELAY "mpeg4audio_tf_core_coder_delay"`
: Parameter for MPEG-4 audio (AAC) decoding.

`#define P_FORMATKEY_MPEG4AUDIO_TF_LSLAYER_LENGTH "mpeg4audio_tf_lslayer_length"`
: Parameter for MPEG-4 audio (AAC) decoding.

`#define P_FORMATKEY_MPEG4AUDIO_TF_PCE "mpeg4audio_tf_pce"`
: Parameter for MPEG-4 audio (AAC) decoding.

`#define P_FORMATKEY_MPEG4VIDEO_VOP_TIME_INC_RESOLUTION "mpeg4video_vop_time_inc_res"`
: MPEG-4 video object plane temporal resolution.

`#define P_FORMATKEY_YCBCR420_PLANAR_VIDEO_UV_STRIDE "ycbcr420_uv_stride"`
: The distance in bytes from one row to the next.

`#define P_FORMATKEY_YCBCR420_PLANAR_VIDEO_Y_STRIDE "ycbcr420_y_stride"`
: The distance in bytes from one row to the next.

fmtAudioChannelUsage Enum

Purpose Values for the key `P_FORMATKEY_CHANNEL_USAGE`.

Declared In `MMFormatDefs.h`

Constants `P_STEREO = 0x02`
: Stereo.

`P_MONO = 0x01`
Mono.

`P_DOLBY_PRO_LOGIC_STEREO = 0x12`
Dolby digital pro logic stereo.

`P_DOLBY_5_1_SURROUND = 0x26`
Dolby digital 5.1 format stereo.

`P_DTS_SURROUND = 0x36`
Sony DTS surround stereo.

`P_CHANNEL_COUNT_MASK = 0x0f`
When you AND a channel usage constant with this mask, you get the actual number of audio channels.

fmtMPEG12AudioChannelMode Enum

Purpose Values for the key `P_FORMATKEY_MPEG12_AUDIO_CHANNEL_MODE`.

Declared In `MMFormatDefs.h`

Constants `P_MPEG12_AUDIO_STEREO = 2`
Stereo (2 channels).

`P_MPEG12_AUDIO_JOINT_STEREO = 0x82`
Joint stereo.

`P_MPEG12_AUDIO_DUAL_CHANNEL = 0x42`
Dual channel.

`P_MPEG12_AUDIO_MONO = 1`
Mono (single channel).

`P_MPEG12_AUDIO_CHANNEL_COUNT_MASK = 0x0f`
Mask of all channel mode values.

fmtMPEG12AudioEmphasis Enum

Purpose For internal use only.

Declared In `MMFormatDefs.h`

Constants `P_MPEG12_AUDIO_EMPHASIS_NONE = 0`

`P_MPEG12_AUDIO_EMPHASIS_50_15ms = 1`

`P_MPEG12_AUDIO_EMPHASIS_CCITT_J17 = 3`

fmtMPEG12AudioLayer Enum

Purpose Possible audio layers implemented in MPEG-1 and MPEG-2.

Declared In `MMFormatDefs.h`

Constants `P_MPEG12_AUDIO_LAYER_I`
Audio layer 1.

`P_MPEG12_AUDIO_LAYER_II`
Audio layer 2.

`P_MPEG12_AUDIO_LAYER_III`
Audio layer 3 (MP3).

fmtMPEG12AudioRevision Enum

Purpose Possible versions when the format is MPEG-1 or MPEG-2.

Declared In `MMFormatDefs.h`

Constants `P_MPEG12_AUDIO_REV_MPEG1`
MPEG-1.

`P_MPEG12_AUDIO_REV_MPEG2`
MPEG-2.

`P_MPEG12_AUDIO_REV_MPEG2_5`
MPEG-2.5.

fmtMPEG4AudioObjectProfile Enum

Purpose Values for the key `P_FORMATKEY_MPEG4AUDIO_OBJECT_PROFILE`.

Declared In `MMFormatDefs.h`

Constants `P_MPEG4AUDIO_AAC_MAIN = 1`
Advanced Audio Coding (AAC) main.

`P_MPEG4AUDIO_AAC_LC`
Low-complexity AAC.

`P_MPEG4AUDIO_AAC_SSR`
Scalable sampling rate.

`P_MPEG4AUDIO_AAC_LTP`
AAC coder including long term prediction.

`P_MPEG4AUDIO_AAC_SCALABLE = 6`
AAC scalable.

`P_MPEG4AUDIO_TWINVQ`
TwinVQ audio profile.

`P_MPEG4AUDIO_CELP`
CELP speech coder.

`P_MPEG4AUDIO_HVXC`
HVXC speech coder.

`P_MPEG4AUDIO_TTSI = 12`
Text-to-speech interface.

`P_MPEG4AUDIO_MAIN_SYNTHETIC`
Synthesis profile.

`P_MPEG4AUDIO_WAVETABLE`
Wavetable synthesis.

`P_MPEG4AUDIO_GENERAL_MIDI`
General MIDI synthesis.

`P_MPEG4AUDIO_ALGORITHMIC`
Algorithmic synthesis.

`P_MPEG4AUDIO_ER_AAC_LC`
Error-resilient low-complexity AAC.

`P_MPEG4AUDIO_ER_AAC_LTP`
Error-resilient AAC coder including long term prediction.

`P_MPEG4AUDIO_ER_AAC_SCALABLE`
Error-resilient AAC scalable.

`P_MPEG4AUDIO_ER_TWINVQ`
Error-resilient TwinVQ audio profile.

`P_MPEG4AUDIO_ER_BSAC`
Error-resilient bit-sliced arithmetic coding.

`P_MPEG4AUDIO_ER_AAC_LD`
Error-resilient low-delay AAC.

`P_MPEG4AUDIO_ER_CELP`
Error-resilient CELP speech coder.

`P_MPEG4AUDIO_ER_HVXC`
Error-resilient HVXC speech coder.

`P_MPEG4AUDIO_ER_HILN`
Error-resilient HILN parametric coding.

`P_MPEG4AUDIO_ER_PARAMETRIC`
Error-resilient parametric coding.

fmtMPEG4AudioTFCoding Enum

Purpose Values for `P_FORMATKEY_MPEG4AUDIO_TF_CODING`.

Declared In `MMFormatDefs.h`

Constants `P_MPEG4AUDIO_TF_AAC_SCALABLE`
AAC scalable.

`P_MPEG4AUDIO_TF_BSAC`
Bit-sliced arithmetic coding.

`P_MPEG4AUDIO_TF_TWINVQ`
Twin VQ.

`P_MPEG4AUDIO_TF_AAC_NON_SCALABLE`
AAC non-scalable.

fmtRawAudioType Enum

Purpose Values for the key `P_FORMATKEY_RAW_AUDIO_TYPE`.

Declared In `MMFormatDefs.h`

Constants `P_AUDIO_INT8 = 0x01`
8-bit integer.

`P_AUDIO_UINT8 = 0x11`
8-bit unsigned integer.

`P_AUDIO_INT16 = 0x02`
16-bit integer.

`P_AUDIO_INT32 = 0x04`
32-bit integer.

`P_AUDIO_FLOAT = 0x24`
Floating-point value.

`P_AUDIO_SIZE_MASK = 0x0f`
: When you AND a raw audio type constant with this mask, you get the actual audio sample size in bytes.

fmtVideoOrientation Enum

Purpose Values for the key `P_FORMATKEY_VIDEO_ORIENTATION`.

Declared In `MMFormatDefs.h`

Constants `P_VIDEO_TOP_LEFT_RIGHT = 1`
: Start at the top and move left to right.

`P_VIDEO_BOTTOM_LEFT_RIGHT`
: Start at the bottom and scan left to right.

formatFamily Enum

Purpose Values of the first byte in a `formatType`.

Declared In `MMFormatDefs.h`

Constants `P_FORMAT_FAMILY_MPEG12 = 1`
: MPEG-1 or MPEG-2 multimedia file.

`P_FORMAT_FAMILY_MPEG4 = 3`
: MPEG-4 multimedia file.

`P_FORMAT_FAMILY_ATRAC`
: Adaptive Transform Acoustic Coding for MiniDisc sound file.

`P_FORMAT_FAMILY_H263`
: H.263 multimedia format.

`P_FORMAT_FAMILY_3GPP`
: 3GPP multimedia format file.

`P_FORMAT_FAMILY_AVI`
: Microsoft's video for Windows standard.

`P_FORMAT_FAMILY_QUICKTIME`
: Apple QuickTime video standard.

`P_FORMAT_FAMILY_ASF`
: Microsoft advanced streaming format multimedia.

`P_FORMAT_FAMILY_WAV`
: WAV format sound files.

`P_FORMAT_FAMILY_AIFF`
: Audio Interchange File Format for sampled sound.

`P_FORMAT_FAMILY_JPEG`
: A JPEG graphic file.

`P_FORMAT_FAMILY_GIF`
: A GIF graphic file.

`P_FORMAT_FAMILY_BMP`
: A BMP graphic file.

`P_FORMAT_FAMILY_PNG`
: A PNG graphic file.

`P_FORMAT_FAMILY_TIFF`
: A TIFF graphic file.

`P_FORMAT_FAMILY_RAW`
: Raw format for audio or video.

`P_FORMAT_FAMILY_YCBCR420`
: YCbCr420 video format.

`P_FORMAT_FAMILY_OGG`
: Ogg Vorbis format.

`P_FORMAT_FAMILY_AMR`
: 3GPP AMR speech format.

`P_FORMAT_FAMILY_G7XX`
: CCITT 7xx voice compression formats.

`P_FORMAT_FAMILY_H264`
: H264 video file.

`P_FORMAT_FAMILY_CINEPAK`
: SuperMac Cinepak video format.

`P_FORMAT_FAMILY_USER = 0xfd`
: User-defined format.

`P_FORMAT_FAMILY_PRIVATE = 0xfe`
: Private format.

`P_FORMAT_FAMILY_PALMOS_INTERNAL = 0xff`
: Internal to Palm OS®.

formatType Enum

Purpose Values of format types.

Declared In `MMFormatDefs.h`

Constants

`P_FORMAT_UNKNOWN = 0`
Unknown format.

`P_FORMAT_ANY_TYPE = 0xff`
Matches any format.

`P_FORMAT_AUDIO_TYPE = 0x01`
Any audio format.

`P_FORMAT_VIDEO_TYPE = 0x02`
Any video format.

`P_FORMAT_MIDI_TYPE = 0x04`
Matches any MIDI sound type.

`P_FORMAT_STILL_TYPE = 0x08`
Matches any still graphic file.

`P_FORMAT_RAW_AUDIO = _MMFORMATTYPE(0, P_FORMAT_FAMILY_RAW, P_FORMAT_AUDIO_TYPE),`
Raw audio format.

`P_FORMAT_RAW_VIDEO = _MMFORMATTYPE(0, P_FORMAT_FAMILY_RAW, P_FORMAT_VIDEO_TYPE)`
Raw video format.

`P_FORMAT_MIDI = _MMFORMATTYPE(0, P_FORMAT_FAMILY_RAW, P_FORMAT_MIDI_TYPE),`
Raw MIDI sound file.

`P_FORMAT_RAW_STILL = _MMFORMATTYPE(0, P_FORMAT_FAMILY_RAW, P_FORMAT_STILL_TYPE)`
Raw graphics file.

`P_FORMAT_MPEG12_AUDIO = _MMFORMATTYPE(0, P_FORMAT_FAMILY_MPEG12, P_FORMAT_AUDIO_TYPE)`
MPEG-1 or MPEG-2 audio track.

`P_FORMAT_MPEG12_VIDEO = _MMFORMATTYPE(0, P_FORMAT_FAMILY_MPEG12, P_FORMAT_VIDEO_TYPE)`
MPEG-1 or MPEG-2 video track.

```
P_FORMAT_MPEG4_AUDIO_TF =
  _MMFORMATTYPE(0x01, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_AUDIO_TYPE)
```
MPEG-4 audio track time/frequency coder.

```
P_FORMAT_MPEG4_AUDIO_CELP =
  _MMFORMATTYPE(0x02, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_AUDIO_TYPE)
```
MPEG-4 audio track code excited linear prediction.

```
P_FORMAT_MPEG4_AUDIO_PARAMETRIC =
  _MMFORMATTYPE(0x03, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_AUDIO_TYPE)
```
MPEG-4 audio track with parametric coding.

```
P_FORMAT_MPEG4_AUDIO_TTS =
  _MMFORMATTYPE(0x04, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_AUDIO_TYPE)
```
MPEG-4 audio text-to-speech track.

```
P_FORMAT_MPEG4_AUDIO_STRUCTURED =
  _MMFORMATTYPE(0x05, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_AUDIO_TYPE)
```
MPEG-4 audio track with structured format.

```
P_FORMAT_MPEG4_VIDEO =
  _MMFORMATTYPE(0, P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_VIDEO_TYPE)
```
MPEG-4 video track.

```
P_FORMAT_ATRAC_AUDIO = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_ATRAC, P_FORMAT_AUDIO_TYPE)
```
ATRAC audio track.

```
P_FORMAT_H263_VIDEO = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_H263, P_FORMAT_VIDEO_TYPE)
```
H.263 video track.

```
P_FORMAT_MSADPCM_AUDIO =
  _MMFORMATTYPE(0x02, P_FORMAT_FAMILY_WAV,
  P_FORMAT_AUDIO_TYPE)
```
A WAV (MS-ADPCM) audio track.

```
P_FORMAT_DVI_ADPCM_AUDIO =
  _MMFORMATTYPE(0x11, P_FORMAT_FAMILY_WAV,
  P_FORMAT_AUDIO_TYPE)
```
Intel/DVI ADPCM audio track.

```
P_FORMAT_JPEG_STILL =  _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_JPEG, P_FORMAT_STILL_TYPE)
```
JPEG graphics file.

```
P_FORMAT_BMP_STILL = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_BMP, P_FORMAT_STILL_TYPE)
```
BMP graphics file.

```
P_FORMAT_PNG_STILL = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_PNG, P_FORMAT_STILL_TYPE)
```
PNG graphics file.

```
P_FORMAT_TIFF_STILL = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_TIFF, P_FORMAT_STILL_TYPE)
```
TIFF graphics file.

```
P_FORMAT_YCBCR420_PLANAR_VIDEO = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_YCBCR420, P_FORMAT_VIDEO_TYPE)
```
Video track with YCbCr420 planar format.

```
P_FORMAT_OGG_VORBIS_AUDIO =
  _MMFORMATTYPE(0x01, P_FORMAT_FAMILY_OGG,
  P_FORMAT_AUDIO_TYPE)
```
Ogg Vorbis audio format track.

```
P_FORMAT_AMR_TS26_071_AUDIO =
  _MMFORMATTYPE(0x01, P_FORMAT_FAMILY_AMR,
  P_FORMAT_AUDIO_TYPE)
```
GSM AMR speech format audio track.

```
P_FORMAT_AMR_TS26_171_AUDIO =
  _MMFORMATTYPE(0x02, P_FORMAT_FAMILY_AMR,
  P_FORMAT_AUDIO_TYPE)
```
AMR wideband speech format audio track.

```
P_FORMAT_G711_AUDIO = _MMFORMATTYPE(0x01,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE)
```
CCITT G711 voice compression audio track.

```
P_FORMAT_G722_AUDIO = _MMFORMATTYPE(0x02,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE)
```
CCITT G722 voice compression audio track.

```
P_FORMAT_G723_AUDIO = _MMFORMATTYPE(0x03,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE)
```
CCITT G723 voice compression audio track.

```
P_FORMAT_G723_1_AUDIO = _MMFORMATTYPE(0x04,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE),
```
CCITT G732.1 voice compression audio track.

```
P_FORMAT_G726_AUDIO = _MMFORMATTYPE(0x05,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE),
```
CCITT G726 voice compression audio track.

```
P_FORMAT_G728_AUDIO = _MMFORMATTYPE(0x06,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE),
```
CCITT G728 voice compression audio track.

```
P_FORMAT_G729_AUDIO = _MMFORMATTYPE(0x07,
  P_FORMAT_FAMILY_G7XX, P_FORMAT_AUDIO_TYPE),
```
CCITT G729 voice compression audio track.

```
P_FORMAT_H264_VIDEO = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_H264, P_FORMAT_VIDEO_TYPE)
```
H264 video track.

```
P_FORMAT_CINEPAK_VIDEO = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_CINEPAK, P_FORMAT_VIDEO_TYPE)
```
Cinepack video track.

```
P_FORMAT_STREAM_TYPE = 0x10
```
Matches any streaming type.

```
P_FORMAT_MPEG12_STREAM =
  _MMFORMATTYPE(0, P_FORMAT_FAMILY_MPEG12,
  P_FORMAT_STREAM_TYPE)
```
An MPEG-1 or MPEG-2 streaming video.

```
P_FORMAT_MPEG4_STREAM =
  _MMFORMATTYPE(0,  P_FORMAT_FAMILY_MPEG4,
  P_FORMAT_STREAM_TYPE)
```
An MPEG-4 streaming video.

```
P_FORMAT_MQV_STREAM = _MMFORMATTYPE(0x01,
  P_FORMAT_FAMILY_MPEG4, P_FORMAT_STREAM_TYPE)
```
An MQV streaming video.

```
P_FORMAT_ATRAC_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_ATRAC, P_FORMAT_STREAM_TYPE)
```
ATRAC streaming video.

```
P_FORMAT_AVI_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_AVI, P_FORMAT_STREAM_TYPE)
```
Microsoft's video for Windows streaming video.

```
P_FORMAT_QUICKTIME_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_QUICKTIME,
  P_FORMAT_STREAM_TYPE)
```
QuickTime streaming video.

```
P_FORMAT_ASF_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_ASF, P_FORMAT_STREAM_TYPE)
```
Advanced streaming format video.

```
P_FORMAT_WAV_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_WAV, P_FORMAT_STREAM_TYPE)
```
MS-ADPCM streaming sound.

```
P_FORMAT_AIFF_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_AIFF, P_FORMAT_STREAM_TYPE)
```
AIFF streaming sound.

```
P_FORMAT_BMP_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_BMP, P_FORMAT_STREAM_TYPE)
```
A BMP streaming video file.

```
P_FORMAT_JPEG_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_JPEG, P_FORMAT_STREAM_TYPE)
```
A JPEG streaming video file.

```
P_FORMAT_PNG_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_PNG, P_FORMAT_STREAM_TYPE)
```
A PNG streaming video file.

```
P_FORMAT_TIFF_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_TIFF, P_FORMAT_STREAM_TYPE)
```
A TIFF streaming video file.

```
P_FORMAT_OGG_VORBIS_STREAM = _MMFORMATTYPE(0,
  P_FORMAT_FAMILY_OGG, P_FORMAT_STREAM_TYPE)
```
Ogg Vorbis streaming sound.

Miscellaneous Constants

Purpose Other constants.

Declared In `MMFormatDefs.h, MMFormat.h`

Constants `#define P_FORMAT_MAX_KEY_LENGTH 64`
Maximum length of a format key.

`#define P_MM_INVALID_FORMAT 0`
Invalid format.

Multimedia Format Functions and Macros

_MMFORMATTYPE Macro

Purpose Defines a value for a media format type like those in the `formatType` enum.

Declared In `MMFormatDefs.h`

Prototype `#define _MMFORMATTYPE(`*`subtype`*`, `*`family`*`, `*`type`*`)`

Parameters → *`subtype`*
An 8-bit integer that describes a family subtype.

→ *`family`*
An 8-bit integer that describes a format family using one of the `formatFamily` constants.

→ *`type`*
An 8-bit integer that describes the basic media type: audio track, video track, or stream. Only 5 bits are currently used. The remaining 3 are reserved for future type expansion.

Returns A 32-bit constant that uniquely identifies a multimedia format.

Comments If you need to define a format type for a proprietary format, use `P_FORMAT_FAMILY_PRIVATE` as the family.

MMFormatCopy Function

Purpose Creates a new media format object by copying an existing media format object.

Declared In `MMFormat.h`

Prototype

```
status_t MMFormatCopy (MMFormat *outDest,
   MMFormat source)
```

Parameters ← *`outDest`*

Pointer to the newly created media format object that is identical to *`source`*.

→ *`source`*

A valid media format object.

Returns `errNone` if the function was successful, otherwise an appropriate Multimedia Library error.

Comments The caller is responsible for calling `MMFormatDelete()` on the newly created media format object when they no longer need it.

MMFormatCreate Function

Purpose Creates a new media format object.

Declared In `MMFormat.h`

Prototype

```
status_t MMFormatCreate (MMFormat *outFormat)
```

Parameters ← *`outFormat`*

Pointer to the newly created media format object.

Returns `errNone` if the function was successful, otherwise an appropriate Multimedia Library error.

Comments The caller is responsible for calling `MMFormatDelete()` on the newly created media format object when they no longer need it.

MMFormatDelete Function

Purpose Deletes a media format object.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatDelete (MMFormat format)
```

Parameters → *`format`*
A valid media format object.

Returns `errNone` if the function was successful, otherwise an appropriate Multimedia Library error.

MMFormatEnumerateTerms Function

Purpose Lists the terms in a media format object.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatEnumerateTerms (MMFormat format,
   int32_t *ioIterator, char *outKey,
   MMTypeCode *outType)
```

Parameters → *`format`*
A valid media format object.

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outKey`*
Pointer to the key for the current term. This must point to a buffer of size `P_FORMAT_MAX_KEY_LENGTH` or larger.

← *`outType`*
Pointer to the data type of the current term. Unspecified wildcard terms may have type `P_MM_TYPE_WILD` (in which case they contain no data).

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
: The specified format object is invalid.

`sysErrBadIndex`
: The iterator value is invalid or past the last item in the set.

MMFormatGetTerm Function

Purpose Returns the value of a term in a media format object.

Declared In `MMFormat.h`

Prototype

```
status_t MMFormatGetTerm (MMFormat format,
   const char *key, MMTypeCode typeCode,
   void *outValue, int32_t *ioLength)
```

Parameters

→ *`format`*
: A valid media format object.

→ *`key`*
: Pointer to the key to search for.

→ *`typeCode`*
: Pointer to the expected type of data. Specify one of the constants listed in "MMTypeCode" on page 73.

↔ *`outValue`*
: On input, a pointer to a buffer capable of storing the specified type of data. On output, a pointer to the value, if it was found and enough storage space was provided.

↔ *`ioLength`*
: On input, if *`typeCode`* specifies a variable-length value with no delimiter (`P_MM_TYPE_RAW`,), then this is a pointer to the size of the *`outValue`* buffer in bytes; otherwise this may be `NULL`. On output, a pointer to the actual size of the value.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The specified format object is invalid.

`sysErrBadIndex`
: The specified key was not found.

`sysErrBadType`
The specified *`typeCode`* does not match the terms's type.

`memErrNotEnoughSpace`
The *`outValue`* buffer was filled to capacity with a partial value and *`ioLength`* is set to indicate the required capacity.

See Also `MMFormatGetTermInt32()`, `MMFormatGetTermType()`, `MMFormatSetTerm()`

MMFormatGetTermInt32 Function

Purpose Returns the value of a term in a media format object as an `int32_t` type.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatGetTermInt32 (MMFormat format,
   const char *key, int32_t *outValue)
```

Parameters

→ *`format`*
A valid media format object.

→ *`key`*
Pointer to the key to search for.

↔ *`outValue`*
On input, a pointer to a buffer capable of storing the requested value. On output, a pointer to the value, if it was found and is representable as an `int32_t` (not a `P_MM_TYPE_WILD`, for example).

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified format object is invalid.

`sysErrBadIndex`
The specified key was not found.

`sysErrBadType`
The terms's value could not be converted to an `int32_t`.

See Also `MMFormatGetTerm()`

MMFormatGetTermType Function

Purpose Returns the type of a term in a media format object.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatGetTermType (MMFormat format,
   const char *key, MMTypeCode *outType)
```

Parameters

→ *format*
A valid media format object.

→ *key*
Pointer to the key to find the type of.

← *outType*
The term type.

Returns The following result codes:

`errNone`
No error.

`sysErrBadIndex`
The specified key was not found.

See Also `MMFormatGetTerm()`

MMFormatGetType Function

Purpose Returns the type of a media format object.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatGetType (MMFormat format,
   MMFormatType *outType)
```

Parameters

→ *format*
A valid media format object.

← *outType*
The media format type.

Returns `errNone` if the function was successful, otherwise an appropriate Multimedia Library error.

See Also `MMFormatSetType()`

MMFormatRawAudio Function

Purpose	Returns the standard media format used to describe raw audio.
Declared In	`MMFormat.h`
Prototype	`MMFormat MMFormatRawAudio (void)`
Parameters	None.
Returns	The standard media format used to describe raw audio.
See Also	`MMFormatRawStill()`, `MMFormatRawVideo()`

MMFormatRawStill Function

Purpose	Returns the standard media format used to describe a raw still image.
Declared In	`MMFormat.h`
Prototype	`MMFormat MMFormatRawStill (void)`
Parameters	None.
Returns	The standard media format used to describe a raw still image.
See Also	`MMFormatRawAudio()`, `MMFormatRawVideo()`

MMFormatRawVideo Function

Purpose	Returns the standard media format used to describe raw video.
Declared In	`MMFormat.h`
Prototype	`MMFormat MMFormatRawVideo (void)`
Parameters	None.
Returns	The standard media format used to describe raw video.
See Also	`MMFormatRawAudio()`, `MMFormatRawStill()`

MMFormatsCompatible Function

Purpose Tests if two media format objects are compatible by matching their types and terms.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatsCompatible (MMFormat a,
    MMFormat b)
```

Parameters

→ *a*
A valid media format object.

→ *b*
A valid media format object.

Returns The following result codes:

`errNone`
The formats are compatible.

`mediaErrFormatMismatch`
The formats are not compatible.

`sysErrParamErr`
One or both of the specified media formats has not been initialized.

MMFormatSetTerm Function

Purpose Sets the value of a term in a media format object, replacing any existing value for the given key.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatSetTerm (MMFormat format,
    const char *key, MMTypeCode typeCode,
    const void *value, int32_t length)
```

Parameters

→ *format*
A valid media format object.

→ *key*
Pointer to the key to set the value for.

→ *typeCode*
The type of data provided in `value`.

→ *`value`*
Pointer to the value to set for the term identified by *`key`*.

→ *`length`*
If *`typeCode`* specifies a variable-length value with no delimiter (such as `P_MM_TYPE_RAW`), this parameter should indicate the size of the value in bytes. Otherwise, this parameter is ignored and should be set to 0.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified media format is invalid.

See Also `MMFormatGetTerm()`

MMFormatSetTermInt32 Function

Purpose Sets the value of a term in a media format object to the specified `int32_t` value, replacing any existing value for the given key.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatSetTermInt32 (MMFormat format,
   const char *key, int32_t value)
```

Parameters

→ *`format`*
A valid media format object.

→ *`key`*
Pointer to the key to set the value for.

→ *`value`*
Pointer to the value to set for the term identified by *`key`*.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified media format is invalid.

See Also `MMFormatGetTermInt32()`

MMFormatSetType Function

Purpose Changes the basic format type of a media format object.

Declared In `MMFormat.h`

Prototype
```
status_t MMFormatSetType (MMFormat format,
    MMFormatType type)
```

Parameters

→ *format*
A valid media format object.

→ *type*
The new type. Specify one of the constants listed in "formatType" on page 93.

Returns `errNone` if the function was successful, otherwise an appropriate Multimedia Library error.

See Also `MMFormatGetType()`

7

Multimedia Properties

This chapter describes multimedia functions related to properties.

The header file `MMProperty.h` declares the API that this chapter describes.

Multimedia Property Functions and Macros

MMPropertyEnumerate Function

Purpose Iterates through the supported values of a particular property of an object.

Declared In `MMProperty.h`

Prototype

```
status_t MMPropertyEnumerate (int32_t id,
    int32_t property, int32_t *ioIterator,
    MMTypeCode typeCode, void *outValue,
    int32_t *ioLength)
```

Parameters

→ *id*
A valid object ID.

→ *property*
A valid property key.

↔ *ioIterator*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

→ *typeCode*
The expected type of data. Specify one of the constants listed in "MMTypeCode" on page 73.

↔ *`outValue`*
On input, a pointer to a buffer capable of storing the specified type of data. On output, a pointer to the value, if it was found and enough storage space was provided.

↔ *`ioLength`*
On input, if *`typeCode`* specifies a variable-length value with no delimiter (`P_MM_TYPE_RAW`,), then this is a pointer to the size of the *`outValue`* buffer in bytes; otherwise this may be `NULL`. On output, a pointer to the actual size of the value.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified object or property is invalid.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

`sysErrBadType`
The specified *`typeCode`* does not match the property's type.

`sysErrNotAllowed`
Cannot enumerate this property.

`memErrNotEnoughSpace`
The *`outValue`* buffer was filled to capacity with a partial value and *`ioLength`* is set to indicate the required capacity.

MMPropertyGet Function

Purpose Returns the value of an object property.

Declared In `MMProperty.h`

Prototype
```
status_t MMPropertyGet (int32_t id,
   int32_t property, MMTypeCode typeCode,
   void *outValue, int32_t *ioValueLen)
```

Parameters → *`id`*
A valid object ID.

→ *`property`*
A valid property key.

→ *typeCode*
: The expected type of data. Specify one of the constants listed in "MMTypeCode" on page 73.

↔ *outValue*
: On input, a pointer to a buffer capable of storing the specified type of data. On output, a pointer to the value, if it was found and enough storage space was provided.

↔ *ioLength*
: On input, if *typeCode* specifies a variable-length value with no delimiter (`P_MM_TYPE_RAW`,), then this is a pointer to the size of the *outValue* buffer in bytes; otherwise this may be `NULL`. On output, a pointer to the actual size of the value.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The specified object or property is invalid.

`sysErrBadType`
: The specified *typeCode* does not match the property's type.

`memErrNotEnoughSpace`
: The *outValue* buffer was filled to capacity with a partial value and *ioLength* is set to indicate the required capacity.

See Also `MMPropertySet()`

MMPropertyInfo Function

Purpose Returns metadata about a property.

Declared In `MMProperty.h`

Prototype

```
status_t MMPropertyInfo (int32_t id,
   MMPropInfoType infoType, int32_t property,
   MMTypeCode typeCode, void *outValue,
   int32_t *ioValueLen)
```

Parameters → *id*
: A valid object ID.

→ *`infoType`*
The type of information to return. Specify one of the constants listed in "MMPropInfoType" on page 72.

→ *`property`*
A valid property key.

→ *`typeCode`*
The expected type of data. Specify one of the constants listed in "MMTypeCode" on page 73.

↔ *`outValue`*
On input, a pointer to a buffer capable of storing the specified type of data. On output, a pointer to the value, if it was found and enough storage space was provided.

↔ *`ioLength`*
On input, if *`typeCode`* specifies a variable-length value with no delimiter (`P_MM_TYPE_RAW`,), then this is a pointer to the size of the *`outValue`* buffer in bytes; otherwise this may be `NULL`. On output, a pointer to the actual size of the value.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified object or property is invalid.

`sysErrBadType`
The specified *`typeCode`* does not match the property's type.

`memErrNotEnoughSpace`
The *`outValue`* buffer was filled to capacity with a partial value and *`ioLength`* is set to indicate the required capacity.

See Also `MMPropertyGet()`

MMPropertySet Function

Purpose Sets an object property value.

Declared In `MMProperty.h`

Prototype
```
status_t MMPropertySet (int32_t id,
    int32_t property, MMTypeCode typeCode,
    const void *value, uint32_t length)
```

Parameters

→ *`id`*
A valid object ID.

→ *`property`*
A valid property key.

→ *`typeCode`*
The expected type of data. Specify one of the constants listed in "MMTypeCode" on page 73.

→ *`value`*
A pointer to the value to set for the property.

→ *`length`*
If *`typeCode`* specifies a variable-length value with no delimiter (`P_MM_TYPE_RAW`,), then this indicates the size of the value in bytes. Otherwise, *`length`* is ignored and should be set to 0.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The specified object or property is invalid.

`sysErrBadType`
The specified *`typeCode`* does not match the property's type.

See Also `MMPropertyGet()`

8 Multimedia Sessions

This chapter describes the session management API in the Multimedia Library. It covers:

The header files `MMSession.h` and `MMSessionClass.h` declare the API that this chapter describes.

Multimedia Session Structures and Types

MMSessionEvent Struct

Purpose Defines a notification that an event has occurred.

Declared In `MMSession.h`

Prototype

```
typedef struct MMSessionEventTag {
   MMSessionID sessionRef;
   MMEvent eventCode;
   int32_t eventCause;
} MMSessionEvent
```

Fields

`sessionRef`
The session’s unique identifier.

`eventCode`
Identifies the event that occurred. This is one of the Session Notifications.

`eventCause`
: Identifies the cause of the event. This is one of the Session Event Causes.

Comments This structure defines the parameter block for the `sysAppLaunchCmdMultimediaEvent` launch code and an event for the `MMSessionCallbackFn()` callback function.

Multimedia Session Constants

Camera Flash Mode Values Enum

Purpose Flash modes for a digital camera, used as a value for the `P_MM_SOURCE_PROP_CAMERA_FLASH_MODE` property key. These constants specify whether the camera uses a flash when taking a picture.

Declared In `MMSession.h`

Constants

`P_MM_FLASH_MODE_OFF`
: Never use flash.

`P_MM_FLASH_MODE_AUTO`
: Automatically detect when a flash is needed.

`P_MM_FLASH_MODE_FRONT`
: Fill flash mode, which illuminates the front part of the image.

`P_MM_FLASH_MODE_SLOW`
: Slow synchronization flash mode, which preserves the background.

`P_MM_FLASH_MODE_REAR`
: Rear flash mode. Flashes at the end of the picture. Useful for action shots.

Camera Focus Values Enum

Purpose Specifies the focus for a camera. Values for the `P_MM_SOURCE_PROP_CAMERA_FOCUS` property key.

Declared In `MMSession.h`

Constants

`P_MM_FOCUS_AUTO = 0`
: Automatically adjust focus.

`P_MM_FOCUS_INFINITY = LONG_MAX`
: Use infinite focus.

Camera Property Key Constants

Purpose Constants used to identify properties within a source that represents a camera. The descriptions below explain the value stored for the key and give the type for the value.

Declared In `MMSession.h`

Constants

```
#define P_MM_SOURCE_PROP_CAMERA_APERTURE
   (P_MM_SOURCE_PROP_BASE | 0x0012L)
```
The camera's aperture as a 32-bit integer.

```
#define P_MM_SOURCE_PROP_CAMERA_EXPOSURE
   (P_MM_SOURCE_PROP_BASE | 0x0010L)
```
A 32-bit integer specifying the reciprocal of the exposure in seconds or the constant `P_MM_EXPOSURE_AUTO`.

```
#define P_MM_SOURCE_PROP_CAMERA_EXPOSURE_SCALE
   (P_MM_SOURCE_PROP_BASE | 0x0011L)
```
A 32-bit integer by which the exposure is modified.

```
#define P_MM_SOURCE_PROP_CAMERA_FLASH_MODE
   (P_MM_SOURCE_PROP_BASE | 0x0013L)
```
One of the Camera Flash Mode Values.

```
#define P_MM_SOURCE_PROP_CAMERA_FOCUS
   (P_MM_SOURCE_PROP_BASE | 0x0016L)
```
One of the Camera Focus Values.

```
#define P_MM_SOURCE_PROP_CAMERA_ISO_SENSITIVITY
   (P_MM_SOURCE_PROP_BASE | 0x0018L)
```
A 32-bit integer specifying an ISO ASA number or the constant `P_MM_ISO_SENSITIVITY_AUTO`.

`#define P_MM_SOURCE_PROP_CAMERA_RED_EYE_REDUCTION`
`(P_MM_SOURCE_PROP_BASE | 0x0014L)`
A Boolean value specifying whether red-eye reduction is on or off.

`#define P_MM_SOURCE_PROP_CAMERA_WHITE_BALANCE`
`(P_MM_SOURCE_PROP_BASE | 0x0015L)`
One of the Camera White Balance Values.

`#define P_MM_SOURCE_PROP_CAMERA_ZOOM`
`(P_MM_SOURCE_PROP_BASE | 0x0017L)`
A 32-bit integer specifying the zoom level.

Camera White Balance Values Enum

Purpose White balance values for digital camera, values for the `P_MM_SOURCE_PROP_CAMERA_WHITE_BALANCE` property key. White balance adjusts the color synchronization for various lighting conditions.

Declared In `MMSession.h`

Constants `P_MM_WHITE_BALANCE_AUTO`
Automatically adjust white balance.

`P_MM_WHITE_BALANCE_INDOOR`
Indoor lighting.

`P_MM_WHITE_BALANCE_OUTDOOR`
Outdoor lighting.

`P_MM_WHITE_BALANCE_FLUORESCENT`
Fluorescent lighting.

Default Session Class IDs

Purpose Used to identify default session classes within the Movie Server.

Declared In `MMSessionClass.h`

Constants `#define P_MM_SESSION_CLASS_DEFAULT_CAPTURE`
`(P_MM_STANDARD_SESSION_CLASS_BASE | 0x0002L)`
The class ID for the default recording (capture) session class.

`#define P_MM_SESSION_CLASS_DEFAULT_PLAYBACK (P_MM_STANDARD_SESSION_CLASS_BASE | 0x0001L)`
The class ID for the default playback session class.

See Also `MMSessionCreate()`

Default URLs

Purpose Default URLs used to instantiate sources or destinations within a session.

Declared In `MMSession.h`

Constants

`#define P_MM_NULL_URL "palmdev:///Media/Null"`
Represents no device.

`#define P_MM_DEFAULT_AUDIO_CAPTURE_URL "palmdev:///Media/Default/AudioIn"`
Any audio recording device.

`#define P_MM_DEFAULT_AUDIO_RENDER_URL "palmdev:///Media/Default/AudioOut"`
Any audio playback device.

`#define P_MM_DEFAULT_STILL_CAPTURE_URL "palmdev:///Media/Default/StillIn"`
Any still capture input (such as a camera).

`#define P_MM_DEFAULT_VIDEO_CAPTURE_URL "palmdev:///Media/Default/VideoIn"`
Any video recording device.

`#define P_MM_DEFAULT_VIDEO_RENDER_URL "palmdev:///Media/Default/VideoOut"`
Any video playback device.

Comments The `CAPTURE_URLs` instantiate a Movie Server object that represents the source of recorded input, and the `RENDER_URLs` instantiate a Movie Server object that represents the destination for recorded material, within the default recording sessions. In playback sessions, the `RENDER_URLs` instantiate a Movie Server object that represents the destination for played content.

See Also `MMSessionAddDest()`, `MMSessionAddSource()`

Destination Property Key Constants

Purpose Constants used to identify properties in a destination object. The descriptions below explain the value for each key and give the type for the value.

Declared In `MMSession.h`

Constants

`#define P_MM_DEST_PROP_FILE_FORMAT (P_MM_DEST_PROP_BASE | 0x0002L)`
An `MMFormat` type specifying the format written to the destination object.

`#define P_MM_DEST_PROP_URL (P_MM_DEST_PROP_BASE | 0x0001L)`
A string representing the URL used to create a destination object.

ISO Sensitivity Value Enum

Purpose Possible value for the camera's ISO sensitivity level.

Declared In `MMSession.h`

Constants

`P_MM_ISO_SENSITIVITY_AUTO = 0`
Automatically detect ISO sensitivity.

Miscellaneous Session Constants

Purpose Other constants defined in `MMSession.h`.

Declared In `MMSession.h`

Constants

`#define P_MM_EXPOSURE_AUTO 0`
Automatically adjust the exposure for a camera. This is a possible value for the property `P_MM_SOURCE_PROP_CAMERA_EXPOSURE`.

`#define P_MM_DEFAULT_DEST 0`
Specifies a default destination device should be instantiated.

`#define P_MM_DEFAULT_SOURCE 0`
Specifies a default source device should be instantiated.

MMSessionControlOpcode Typedef

Purpose Session control operation codes. A multimedia client application sends these to the session object to have the session perform an action.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMSessionControlOpcode`

Constants

`P_MM_SESSION_CTL_RUN = 0x01`
Start or continue recording or playback.

`P_MM_SESSION_CTL_PAUSE = 0x02`
Pause recording or playback.

`P_MM_SESSION_CTL_STOP = 0x03`
Stop recording or playback.

`P_MM_SESSION_CTL_PREFETCH = 0x04`
Begin buffering data from the source.

`P_MM_SESSION_CTL_GRAB = 0x05`
Grab a still image from a video playback or recording session.

`P_MM_SESSION_CTL_REFRESH = 0x06`
Refresh the display.

`P_MM_SESSION_CTL_CUSTOM_BASE = 0x1000`
Base value where a licensee can add its own control op codes.

See Also `MMSessionControl()`

MMSessionState Typedef

Purpose Constants that describe the session object state.

Declared In `MMDefs.h`

Prototype `typedef int32_t MMSessionState`

Constants

`P_MM_SESSION_NOT_INITIALIZED = 0x01`
The session exists but has not been initialized with the information it needs to playback or record.

`P_MM_SESSION_READY = 0x02`
The session has been initialized and is ready to begin. `MMSessionFinalize()` puts the session in this state.

`P_MM_SESSION_PREFETCHING = 0x03`
The session is buffering data.

`P_MM_SESSION_PAUSED = 0x04`
The session has been paused.

`P_MM_SESSION_RUNNING = 0x05`
The session has begun.

See Also `MMSessionGetState()`

Session Class Constants Enum

Purpose Defines values for the high-order 16 bits of a session class ID.

Declared In `MMSessionClass.h`

Constants `P_MM_STANDARD_SESSION_CLASS_BASE = 0x10010000L`
The high-order bits for a PalmSource-defined session class ID.

`P_MM_USER_SESSION_CLASS_BASE = 0x10020000L`
The high-order bits for a licensee-defined session class ID.

`P_MM_PRIVATE_SESSION_CLASS_BASE = 0x10030000L`
A private session class ID defined by either PalmSource or one of its licensees.

Comments The function `MMSessionClassEnumerate()` returns a list of all standard and user sessions. It never enumerates the private classes. Declaring a session class as private is useful if it is tightly integrated with the application that uses it.

Session Class Properties

Purpose Defines property keys that can be used to access information about a session class.

Declared In `MMSessionClass.h`

```
#define P_MM_SESSION_CLASS_PROP_CREATOR
  (P_MM_SESSION_CLASS_PROP_BASE | 0x0003L)
```
The session class's creator ID.

`#define P_MM_SESSION_CLASS_PROP_NAME`
`(P_MM_SESSION_CLASS_PROP_BASE | 0x0001L)`
The session's name.

`#define P_MM_SESSION_CLASS_PROP_VERSION`
`(P_MM_SESSION_CLASS_PROP_BASE | 0x0002L)`
The session's version number.

Session Creation Constants Enum

Purpose Used in the `MMSessionCreate()` function to determine where to create the session.

Declared In `MMSession.h`

Constants `P_MM_SESSION_CREATE_ANY_PROCESS`
Creates a session in any process. The Movie Server typically creates the session in the Background process.

`P_MM_SESSION_CREATE_LOCAL_PROCESS`
Creates a session in the local process.

Session Default Property Key Constants

Purpose Property keys for session default properties. These are passed to a track when a track is added to a session. The descriptions below explain the value for each key and give the type for the value.

Declared In `MMSession.h`

Constants `#define P_MM_SESSION_DEFAULT_AUDIO_ENABLE`
`(P_MM_SESSION_PROP_BASE | 0x0020L)`
Boolean that specifies whether an audio track is enabled or disabled.

`#define P_MM_SESSION_DEFAULT_AUDIO_VOLUME`
`(P_MM_SESSION_PROP_BASE | 0x0024L)`
A 32-bit integer specifying the volume level.

`#define P_MM_SESSION_DEFAULT_DEST_RECT`
`(P_MM_SESSION_PROP_BASE | 0x0023L)`
The default destination rectangle to use for a video frame. The rectangle specifies both the position and the size.

`#define P_MM_SESSION_DEFAULT_SOURCE_RECT (P_MM_SESSION_PROP_BASE | 0x0022L)`
The default source rectangle to use for a video frame. The rectangle specifies both the position and the size.

`#define P_MM_SESSION_DEFAULT_VIDEO_ENABLE (P_MM_SESSION_PROP_BASE | 0x0021L)`
A Boolean value specifying whether a video track should be enabled.

Session Property Key Constants

Purpose Constants used to identify properties stored in a session. The descriptions below explain the value for each key and give the type for the value.

Declared In `MMSession.h`

Constants

`#define P_MM_SESSION_PROP_CURRENT_TIME (P_MM_SESSION_PROP_BASE | 0x0002L)`
Current recording or playback position given in nanoseconds. The value is a 64-bit signed integer.

`#define P_MM_SESSION_PROP_END_TIME (P_MM_SESSION_PROP_BASE | 0x0008L)`
Ending playback position given in nanoseconds. The value is a 64-bit signed integer.

`#define P_MM_SESSION_PROP_IS_LOCAL (P_MM_SESSION_PROP_BASE | 0x0005L)`
A Boolean specifying if the session is running local to the application process.

`#define P_MM_SESSION_PROP_MARKER (P_MM_SESSION_PROP_BASE | 0x0006L)`
A `nsecs_t` value specifying a marker within a track.

`#define P_MM_SESSION_PROP_PLAYBACK_RATE (P_MM_SESSION_PROP_BASE | 0x0003L)`
A 16-bit integer specifying the audio playback rate, where 1 specifies normal speed, 2 double speed, and so on.

`#define P_MM_SESSION_PROP_PREFETCH_TIME (P_MM_SESSION_PROP_BASE | 0x0004L)`
The amount of data, given in nanoseconds, to buffer.

```
#define P_MM_SESSION_PROP_PUBLIC
   (P_MM_SESSION_PROP_BASE | 0x0001L)
```
A Boolean value of `true` if the session is public or `false` if it is private.

```
#define P_MM_SESSION_PROP_REPEAT_ENABLE
   (P_MM_SESSION_PROP_BASE | 0x0009L)
```
A Boolean that indicates that playback should repeat when the end time is reached.

```
#define P_MM_SESSION_PROP_SESSION_CLASS
   (P_MM_SESSION_PROP_BASE | 0x000AL)
```
A 32-bit integer ID of the session class used.

```
#define P_MM_SESSION_PROP_START_TIME
   (P_MM_SESSION_PROP_BASE | 0x0007L)
```
Starting playback position given in nanoseconds. The value is a 64-bit signed integer.

Source Property Key Constants

Purpose Constants used to identify properties within a source. The descriptions below explain the value for each key and give the type for the value.

Declared In `MMSession.h`

```
#define P_MM_SOURCE_PROP_FILE_FORMAT
   (P_MM_SOURCE_PROP_BASE | 0x0002L)
```
An `MMFormat` specifying the format that the source produces.

```
#define P_MM_SOURCE_PROP_URL
   (P_MM_SOURCE_PROP_BASE | 0x0001L)
```
A string representing the URL used to create a source.

Set to the empty string.

Stream Content Keys

Purpose Keys set for an audio or video stream property set.

Declared In `MMSession.h`

Constants

```
#define P_MM_CONTENT_PROP_ALBUM
  (P_MM_CONTENT_PROP_BASE | 0x0006L)
```
String containing the album or CD for a track.

```
#define P_MM_CONTENT_PROP_ARTIST
  (P_MM_CONTENT_PROP_BASE | 0x0003L)
```
String containing the artist that recorded a track.

```
#define P_MM_CONTENT_PROP_DURATION
  (P_MM_CONTENT_PROP_BASE | 0x0001L)
```
An `int32_t` describing the length of a track in milliseconds.

```
#define P_MM_CONTENT_PROP_GENRE
  (P_MM_CONTENT_PROP_BASE | 0x0005L)
```
String containing the genre of a track.

```
#define P_MM_CONTENT_PROP_PLAYLIST
  (P_MM_CONTENT_PROP_BASE | 0x0004L)
```
String containing the play list for a track.

```
#define P_MM_CONTENT_PROP_TITLE
  (P_MM_CONTENT_PROP_BASE | 0x0002L)
```
String containing the title of a track.

```
#define P_MM_CONTENT_PROP_TRACK_NUMBER
  (P_MM_CONTENT_PROP_BASE | 0x0007L)
```
An `int32_t` giving the track number within the album or CD.

Comments Both playback and recording sessions in the Movie Server set these keys, and they are written to encoded output. The descriptions explain the value for each key and give the type for the value.

A given codec may not support all of these properties. Currently, only the PalmSource MPEG audio extractor supports these properties, but other third-party codecs may also support them.

Stream Property Key Constants

Purpose Constants used to identify properties in a stream object. The descriptions below explain the value for each key and give the type for the value.

Declared In `MMSession.h`

Constants

```
#define P_MM_STREAM_PROP_FORMAT
   (P_MM_STREAM_PROP_BASE | 0x0001L)
```
An `MMFormat` specifying the stream's format.

```
#define P_MM_STREAM_PROP_IS_PREVIEW
   (P_MM_STREAM_PROP_BASE | 0x0003L)
```
A Boolean value that, if `true`, identifies the stream as a preview stream.

```
#define P_MM_STREAM_PROP_LANGUAGE
   (P_MM_STREAM_PROP_BASE | 0x0002L)
```
A string specifying the language of the stream.

Multimedia Session Launch Codes

sysAppLaunchCmdMultimediaEvent

Purpose Sent when multimedia session events occur.

Declared In `CmnLaunchCodes.h`

Prototype `#define sysAppLaunchCmdMultimediaEvent 63`

Parameters The launch code's parameter block pointer references a `MMSessionEvent` structure.

Comments Some applications may need to be informed of events relating to a persistent session, even when the application is no longer running. You can use the function `MMSessionRegisterLaunch()` to register an application to be associated with a session. When session events occur, that application is sublaunched with this launch command.

Multimedia Session Functions and Macros

MMDestEnumerateStreams Function

Purpose Enumerates the streams available in a destination object.

Declared In `MMSession.h`

Prototype

```
status_t MMDestEnumerateStreams (MMDestID dest,
   int32_t *ioIterator, MMStreamID *outStream)
```

Parameters

→ *`dest`*
A valid multimedia destination object ID.

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outStream`*
Pointer to the ID of the next stream in the set. The stream ID remains valid until the session is deleted or `MMSessionRemoveAll()` is called.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
One of input parameters is invalid.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

Comments Before calling this function, the destination object must be finalized by calling `MMDestFinalize()`.

MMDestFinalize Function

Purpose Opens the given destination and creates streams.

Declared In `MMSession.h`

Prototype `status_t MMDestFinalize (MMDestID` *`dest`*`)`

Parameters → *`dest`*
A valid multimedia destination object ID.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The destination object is invalid.

Other multimedia errors can also be returned.

Comments On success, the streams may be enumerated with `MMDestEnumerateStreams()`.

MMSessionAcquireOwnership Function

Purpose Makes the calling process acquire ownership of a session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionAcquireOwnership
    (MMSessionID session)
```

Parameters → *`session`*
A valid multimedia session ID, on which `MMSessionReleaseOwnership()` was previously called.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

`sysErrNotAllowed`
The session is owned by another process.

Comments The session must not be owned by another process.

The session will be deleted automatically when the acquiring process exits.

MMSessionAddDefaultTracks Function

Purpose Adds all tracks applicable to this session, using the given source and/or destination.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionAddDefaultTracks
   (MMSessionID session, MMSourceID source,
   MMDestID dest)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`source`*
A valid multimedia source ID, or `P_MM_DEFAULT_SOURCE`.

→ *`dest`*
A valid multimedia destination ID, or `P_MM_DEFAULT_DEST`.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
One of the sessions is invalid.

Comments A playback session can use this function to add all of the tracks that it can play concurrently.

After a call to this function succeeds, the caller can use `MMSessionEnumerateTracks()` to inspect the created tracks and further configure them prior to calling `MMSessionFinalize()`.

If `P_MM_DEFAULT_SOURCE` or `P_MM_DEFAULT_DEST` are passed for source or destination, sources and/or destinations will be added and finalized as necessary. The caller can use `MMSessionEnumerateSources()` and

MMSessionEnumerateDests() to retrieve source and destination IDs for those objects.

See Also MMSessionAddTrack(), MMSessionRemoveTracks()

MMSessionAddDest Function

Purpose Adds a data destination to the session.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionAddDest (MMSessionID session,
   const char *destURL, MMDestID *outDest)
```

Parameters

→ *session*
A valid multimedia session ID.

→ *destURL*
Pointer to a URL of the destination to add to the session.

← *outDest*
Pointer to a valid multimedia destination ID, if the function succeeds.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session or the URL is invalid.

`sysErrUnsupported`
No more destinations may be added to this session.

Comments Some session classes may expose additional destination properties that can be set before calling MMDestFinalize() to open the destination and create streams.

See Also MMSessionEnumerateDests()

MMSessionAddSource Function

Purpose Adds a data source to the session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionAddSource (MMSessionID session,
   const char *sourceURL, MMSourceID *outSource)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`sourceURL`*
Pointer to a URL of the data source to add to the session.

← *`outSource`*
Pointer to the multimedia source ID, if the function succeeds.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session or the URL is invalid.

`sysErrUnsupported`
No more source may be added to this session.

Comments Some session classes may expose additional source properties that can be set before calling `MMSourceFinalize()` to open the source and create streams.

See Also `MMSessionEnumerateSources()`

MMSessionAddTrack Function

Purpose Adds a track to the session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionAddTrack (MMSessionID session,
   MMStreamID sourceStream,
   MMFormat sourceFormat, MMStreamID destStream,
   MMFormat destFormat, MMTrackID *outTrack)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`sourceStream`*
A valid multimedia source ID.

→ *`sourceFormat`*
A valid source media format ID, or `P_MM_INVALID_FORMAT`.

→ *`destStream`*
A valid multimedia destination ID.

→ *`destFormat`*
A valid destination media format ID, or `P_MM_INVALID_FORMAT`.

← *`outTrack`*
Pointer to the multimedia track ID for the created track, if the function succeeds.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
One of the parameters is invalid.

Comments A track represents a particular data-processing route, which, depending on the session class, may be used for playback (rendering) or capture (storage to a local file or network stream).

See Also `MMSessionAddDefaultTracks()`, `MMSessionEnumerateTracks()`, `MMSessionRemoveTracks()`

MMSessionClassEnumerate Function

Purpose Iterates through the available session classes.

Declared In `MMSessionClass.h`

Prototype
```
status_t MMSessionClassEnumerate
    (int32_t *ioIterator,
    MMSessionClassID *outSessionClassID)
```

Parameters ↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to

`P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outSessionClassID`*
Pointer to the ID of the next available session class.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The iterator is invalid.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

MMSessionControl Function

Purpose Sends control opcodes to the Movie Server for playback, capture, and preview.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionControl (MMSessionID session,
   MMSessionControlOpcode sessionCtl)
```

Parameters → *`session`*
A valid multimedia session ID.

→ *`sessionCtl`*
The opcode. Specify one of the constants listed in "MMSessionControlOpcode" on page 121.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

`sysErrNotAllowed`
Operation not allowed for this session.

See Also `MMSessionSeek()`

MMSessionCreate Function

Purpose Creates a new session.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionCreate
    (MMSessionClassID sessionClass, int32_t flags,
    MMSessionID *outSession)
```

Parameters

→ *`sessionClass`*
A valid multimedia session class ID. Specify one of the constants listed in "Default Session Class IDs" on page 118.

→ *`flags`*
Flag that determines in what process to create the session. Specify one of the constants listed in "Session Creation Constants" on page 123.

↔ *`outSession`*
A valid multimedia session ID, if the function succeeds. On input, this must be 0.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The *`outSession`* parameter is not 0 on input.

`sysErrNoFreeResource`
Operation not allowed for this session.

See Also `MMSessionDelete()`

MMSessionDelete Function

Purpose Deletes a session.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionDelete (MMSessionID session)
```

Parameters

→ *`session`*
A valid multimedia session ID.

Returns The following result codes:

`errNone`
No error.

`sysErrNoInit`
The session is not initialized.

Comments On success, *`session`* will no longer be valid.

See Also `MMSessionCreate()`

MMSessionEnumerate Function

Purpose Iterates through the current public sessions (only the sessions for which the value of `P_MM_SESSION_PROP_PUBLIC` is nonzero).

Declared In `MMSession.h`

Prototype
```
status_t MMSessionEnumerate (int32_t *ioIterator,
   MMSessionID *outSession)
```

Parameters

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outSession`*
Pointer to a valid multimedia session ID for the next session in the set.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The iterator value is 0.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

MMSessionEnumerateDests Function

Purpose Iterates through the destinations in this session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionEnumerateDests
    (MMSessionID session, int32_t *ioIterator,
    MMDestID *outDest)
```

Parameters

→ *session*
A valid multimedia session ID.

↔ *ioIterator*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *outDest*
Pointer to the next destination in the set. This destination ID will remain valid until the session is deleted or `MMSessionRemoveAll()` is called.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid or the iterator value is 0.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

MMSessionEnumerateSources Function

Purpose Iterates through the sources in this session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionEnumerateSources
    (MMSessionID session, int32_t *ioIterator,
    MMSourceID *outSource)
```

Parameters

→ *session*
A valid multimedia session ID.

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outSource`*
Pointer to the next source in the set. This source ID will remain valid until the session is deleted or `MMSessionRemoveAll()` is called.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid or the iterator value is 0.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

MMSessionEnumerateTracks Function

Purpose Iterates through the tracks in this session.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionEnumerateTracks
   (MMSessionID session, int32_t *ioIterator,
   MMTrackID *outTrack)
```

Parameters

→ *`session`*
A valid multimedia session ID.

↔ *`ioIterator`*
Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outTrack`*
Pointer to the next track in the set. This track ID will remain valid until the session is deleted or `MMSessionRemoveAll()` is called.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid or the iterator value is 0.

`sysErrBadIndex`
The iterator value is invalid or past the last item in the set.

MMSessionFinalize Function

Purpose Finalize the set of tracks for this session.

Declared In `MMSession.h`

Prototype `status_t MMSessionFinalize (MMSessionID session)`

Parameters → *`session`*
A valid multimedia session ID.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

`sysErrBadIndex`
One or more tracks depends on a source or destination which could not be added.

Comments Calling this function is the final step in preparing a session to run; after this call succeeds, the session enters the `P_MM_SESSION_READY` state and you may call `MMSessionControl()` to begin playback or capture.

After `MMSessionFinalize()` is called, the following functions may not be called for the session: `MMSessionAddDest()`, `MMSessionAddSource()`, `MMSessionAddTrack()`, and `MMSessionAddDefaultTracks()`.

MMSessionGetState Function

Purpose Returns the current state of a session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionGetState (MMSessionID session,
   MMSessionState *outState)
```

Parameters

→ *`session`*
A valid multimedia session ID.

↔ *`outState`*
The current state of the session. One of the constants listed in "MMSessionState" on page 121.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid or *`outState`* is not 0 on input.

MMSessionRegisterCallback Function

Purpose Registers a callback function to monitor this session by handling events.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionRegisterCallback
   (MMSessionID session,
   MMSessionCallbackFn callback, void *userdata,
   uint32_t eventFlags)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`callback`*
Pointer to the callback function (for details, see `MMSessionCallbackFn()`).

→ *`userdata`*
Pointer to arbitrary user-provided data, or `NULL`. This pointer is passed to the callback function.

→ *`eventFlags`*
Unused; must be set to 0.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
One of the parameters is invalid.

`sysErrNotAllowed`
There are too many callbacks registered for this session.

Comments The Multimedia Library calls your function asynchronously, meaning that it's safe to make other multimedia calls from your function.

You can register multiple callback functions for one session.

See Also `MMSessionUnregisterCallback()`

MMSessionRegisterLaunch Function

Purpose Registers a handler application to be sublaunched when a multimedia event occurs.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionRegisterLaunch
   (MMSessionID session, DatabaseID dbID,
   uint32_t eventFlags)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`dbID`*
Application resource database ID.

→ *`eventFlags`*
Unused; must be set to 0.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
One of the parameters is invalid.

Comments The registered application will be sublaunched with the launch command `sysAppLaunchCmdMultimediaEvent`, with a pointer to a `MMSessionEvent`.

See Also `MMSessionUnregisterLaunch()`

MMSessionReleaseOwnership Function

Purpose Makes a multimedia session persist after the calling process exits.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionReleaseOwnership
   (MMSessionID session)
```

Parameters → *session*
A valid multimedia session ID, created by or owned by the current process.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

Comments When a session is created, and the process that created it exits, the session is automatically deleted. Call this function to make the session persist after the calling process exits. The session will persist in the background, where it is owned by the Movie Server, until it is explicitly deleted or `MMSessionAcquireOwnership()` is called.

MMSessionRemoveAll Function

Purpose Removes from a session all sources, destinations, and the tracks that connect them.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionRemoveAll (MMSessionID session)
```

Parameters → *session*
A valid multimedia session ID.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

Comments This function resets the session state to `P_MM_SESSION_NOT_INITIALIZED`. Non-content properties remain unchanged.

MMSessionRemoveTracks Function

Purpose Removes all tracks.

Declared In `MMSession.h`

Prototype

```
status_t MMSessionRemoveTracks
    (MMSessionID session)
```

Parameters → *`session`*
A valid multimedia session ID.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

Comments This function resets the session state to remove sources, destinations, and the tracks that connect them.

Existing source and destination streams may be used to create new tracks.

This function resets the session state to `P_MM_SESSION_NOT_INITIALIZED`. Non-content properties remain unchanged.

See Also `MMSessionAddTrack()`

MMSessionSeek Function

Purpose Seeks to a different location in the current session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionSeek (MMSessionID session,
   MMSeekOrigin origin, int64_t position)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`origin`*
Point to seek from; the seek location is measured from this point. Specify one of the constants listed in "MMSeekOrigin" on page 73.

→ *`position`*
The distance to seek, in nanoseconds. A positive value means to seek forward and a negative value means to seek backward.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

`sysErrOutOfRange`
`position` is out of range.

`sysErrUnsupported`
Seeking is not supported in this session; some streaming types don't support seeking, or it is a capture session.

Comments This function skips forward or backward in the media.

After a successful seek operation the session is stopped (it enters the `P_MM_SESSION_READY` state), whether or not it was playing previous to the call.

See Also `MMSessionControl()`

MMSessionUnregisterCallback Function

Purpose Unregisters a callback function for a session.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionUnregisterCallback
   (MMSessionID session,
   MMSessionCallbackFn callback, void *userdata)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`callback`*
Pointer to a callback function that was previously registered by `MMSessionRegisterCallback()`.

→ *`userdata`*
Unused.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The session is invalid.

`sysErrBadData`
The specified callback function was not found.

MMSessionUnregisterLaunch Function

Purpose Unregisters a registered event handler.

Declared In `MMSession.h`

Prototype
```
status_t MMSessionUnregisterLaunch
   (MMSessionID session, DatabaseID dbID)
```

Parameters

→ *`session`*
A valid multimedia session ID.

→ *`dbID`*
Application resource database ID that was previously registered by `MMSessionRegisterLaunch()`.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The session is invalid.

`sysErrBadData`
: A matching registration was not found.

MMSourceEnumerateStreams Function

Purpose Iterates through the streams available in a source.

Declared In `MMSession.h`

Prototype
```
status_t MMSourceEnumerateStreams
   (MMSourceID source, int32_t *ioIterator,
   MMStreamID *outStream)
```

Parameters

→ *`source`*
: A valid source ID.

↔ *`ioIterator`*
: Pointer to the value returned by the previous call to this function. On the first call to this function, set this value to `P_MM_ENUM_BEGIN`. When the set of iterated values is exhausted, this function sets this value to `P_MM_ENUM_END`.

← *`outStream`*
: Pointer to the next stream in the set. This stream ID will remain valid until the session is deleted or `MMSessionRemoveAll()` is called.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The session is invalid or the iterator value is 0.

`sysErrBadIndex`
: The iterator value is invalid or past the last item in the set.

Comments The source must be finalized for this call to succeed (`MMSourceFinalize()` must have been called).

MMSourceFinalize Function

Purpose Opens the given data source and creates streams.

Declared In `MMSession.h`

Prototype `status_t MMSourceFinalize (MMSourceID` *`source`*`)`

Parameters → *`source`*
A valid multimedia source ID.

Returns The following result codes:

`errNone`
No error.

`sysErrParamErr`
The source is invalid.

Comments On success, the streams may be enumerated by `MMSourceEnumerateStreams()`.

Application-Defined Functions

MMSessionCallbackFn Function

Purpose Called when multimedia events occur.

Declared In `MMSession.h`

Prototype

```
void (*MMSessionCallbackFn)
    (const MMSessionEvent *event, void *userdata)
```

Parameters → *`event`*
A pointer to a multimedia event; see `MMSessionEvent`.

→ *`userdata`*
A pointer to the user data block passed to `MMSessionRegisterCallback()` when the callback function was registered.

Returns Nothing.

Comments To register a callback function, call `MMSessionRegisterCallback()`.

9

Multimedia Tracks

This chapter describes multimedia structures, constants, and functions related to tracks:

The header file `MMTrack.h` declares the API that this chapter describes.

Multimedia Track Structures and Types

FilterCallbackInfo Struct

Purpose Holds information about a buffer of data for which `MMFilterCallbackFn()` has been called.

Declared In `MMTrack.h`

Prototype

```
typedef struct _FilterCallbackInfo {
   int64_t timeStamp;
   size_t bufferSize;
} FilterCallbackInfo
```

Fields `timeStamp`

Timestamp of the data in nanoseconds. For a playback session, this indicates the actual position in the file, counting from 0 at the start.

Timestamps in a capture session are generated by the hardware driver and may be continuously incrementing even when no data is being captured. In this case, you should treat the value received in the first callback as the “base” value, and measure offsets from there.

`bufferSize`
Size of the buffer passed to the callback function, in bytes.

Multimedia Track Constants

Track Property Key Constants

Purpose Values used to retrieve track object properties.

Declared In `MMTrack.h`

Constants

```
#define P_MM_TRACK_PROP_CODEC_CLASS
  (P_MM_TRACK_PROP_BASE | 0x0008L)
```
The 32-bit ID of the class used for encoding or decoding data on the track.

```
#define P_MM_TRACK_PROP_CURRENT_TIME
  (P_MM_TRACK_PROP_BASE | 0x0009L)
```
The current location in the track stream identified as a time value in nanoseconds.

```
#define P_MM_TRACK_PROP_DEST (P_MM_TRACK_PROP_BASE
  | 0x0006L)
```
ID of the destination object associated with track’s destination stream.

```
#define P_MM_TRACK_PROP_DEST_FORMAT
  (P_MM_TRACK_PROP_BASE | 0x0002L)
```
A string specifying the track’s destination format.

```
#define P_MM_TRACK_PROP_DEST_RECT
  (P_MM_TRACK_PROP_BASE | 0x0031L)
```
The region of the screen to be drawn in native screen pixels.

```
#define P_MM_TRACK_PROP_DEST_STREAM
  (P_MM_TRACK_PROP_BASE | 0x0007L)
```
ID of the stream object associated with the track’s destination.

`#define P_MM_TRACK_PROP_ENABLE (P_MM_TRACK_PROP_BASE | 0x0003L)`
Boolean that describes whether the track is enabled or disabled.

`#define P_MM_TRACK_PROP_SOURCE (P_MM_TRACK_PROP_BASE | 0x0004L)`
ID of the source object associated with the track.

`#define P_MM_TRACK_PROP_SOURCE_FORMAT (P_MM_TRACK_PROP_BASE | 0x0001L)`
A string specifying the track's source format.

`#define P_MM_TRACK_PROP_SOURCE_RECT (P_MM_TRACK_PROP_BASE | 0x0030L)`
The region of the source buffer to be displayed in native screen pixels.

`#define P_MM_TRACK_PROP_SOURCE_STREAM (P_MM_TRACK_PROP_BASE | 0x0005L)`
ID of the stream object associated with the track's source.

`#define P_MM_TRACK_PROP_VOLUME (P_MM_TRACK_PROP_BASE | 0x0020L)`
An integer from 0 to 1024 specifying the current volume level.

Multimedia Track Functions and Macros

MMTrackInsertCallbackFilter Function

Purpose Registers a callback function to process data for a track.

Declared In `MMTrack.h`

Prototype
```
status_t MMTrackInsertCallbackFilter
    (MMTrackID track, MMFilterCallbackFn callback,
    void *userdata)
```

Parameters
→ *track*
A valid track ID.

→ *`callback`*
: Pointer to the callback function (for details, see `MMFilterCallbackFn()`).

→ *`userdata`*
: Pointer to arbitrary user-provided data, or `NULL`. This pointer is passed to the callback function.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: One of the parameters is invalid.

`sysErrNotAllowed`
: There is already a callback registered for this track.

Comments Only one callback may be installed for a track.

See Also `MMTrackRemoveCallbackFilter()`

MMTrackRemoveCallbackFilter Function

Purpose Unregisters a callback function for a track.

Declared In `MMTrack.h`

Prototype
```
status_t MMTrackRemoveCallbackFilter
    (MMTrackID track)
```

Parameters → *`track`*
: A valid track ID for which a callback function has been registered by `MMTrackInsertCallbackFilter()`.

Returns The following result codes:

`errNone`
: No error.

`sysErrParamErr`
: The track is invalid.

`sysErrBadData`
: The specified callback function was not found.

Application-Defined Functions

MMFilterCallbackFn Function

Purpose Called when the track receives a buffer of data.

Declared In `MMTrack.h`

Prototype

```
void (*MMFilterCallbackFn) ( MMTrackID track,
   void *buffer, FilterCallbackInfo *info,
   void *userdata)
```

Parameters

→ *`track`*
Track ID of the track for which the callback function is registered.

↔ *`buffer`*
Pointer to a buffer of track data.

→ *`info`*
Pointer to a `FilterCallbackInfo` structure.

→ *`userdata`*
A pointer to the user data block passed to `MMTrackInsertCallbackFilter()` when the callback function was registered.

Returns Nothing.

Comments This function allows an application to customize functionality by processing track data somehow.

To register a filter callback function, call `MMTrackInsertCallbackFilter()`.

Glossary

AC97	Audio codec 97, an open standard defined by Intel and popular with many hardware manufacturers.
ADC	Analog to digital converter (audio recording).
ADPCM	Adaptive Differential Pulse Code Modulation. A form of PCM that produces a digital signal with a lower bit rate than standard PCM.
Audio sample	A single number representing the amplitude of a waveform at a particular time.
Buffer	A storage area for data.
Channel	An audio stream may consist of multiple interleaved channels. A mono stream has one channel, and a stereo stream has two channels. There is one sample per channel.
DAC	Digital to analog converter (audio playback).
Decoder	Converts a particular encoded data format, such as MS-ADPCM or MPEG-1 video into a format that the output device can understand.
Encoder	Converts one multimedia format (typically a raw format) to another encoded format for the purposes of storing that format.
Frame	For audio, a frame consists of interleaved audio samples (one sample per channel) that are output during one quantized time unit. For video, a frame consists of an entire picture.
Media time	The temporal position within media data.
MMLibrary	A shared library included in the SDK that allows multimedia clients to access multimedia features provided by the Movie Server.

Movie Server	A server that runs in the System process and provides all multimedia functionality.
Multimedia client	An application running in the Application process, that accesses the Movie Server through the MMLibrary. A media player is an example of such an application.
Pan	Stereo balance between left and right speakers.
Performance time	The time as specified by an external time source while media data is recorded or played.
PCM	Pulse Code Modulation. A sampling technique for digitizing analog signals.
Property sets	Objects that expose configurable parameters, or properties, which control the object's behavior. Many of the objects in the Multimedia Subsystem are property sets that allow the client application to configure them.
Session	A session provides a context for an application's media playback or recording tasks in the Movie Server.
Stream	In the audio driver, a sequence of stereo sample pairs. In the Movie Server library, an object that defines the media format handled by a source or destination device.
Track	A route for media data from a source device to a destination device.
Track callback filter	A function provided in the multimedia client that receives buffers of data directly from the Movie Server so that the callback can perform whatever tasks it wants with the data, such as storing it locally, modifying it, etc.

Index